Programming Assignment 2 - Analysis
Kevin Feng Yong Chen
Algorithm for Bioinformatics

Description of the program

This program studies the ramifications of different hashing schemes and collision handling methods. The hash table was implemented as a separated class. 2 hashing functions, division and multiplication, were implemented as the class methods of the hash table class. The 3 collision handling methods (i.e. linear probing, quadratic probing and chaining) were also implemented as class methods of the hash table class. The hash table itself was implemented using the list data structure in python. In the case of scheme 7 and 8, in which the bucket size is 3, the table is implemented as a list of lists where each of the slot in the table is a list with a capacity of 3. In addition, a stack object was used in the hash table class to keep track of the free space used in the schemes that involved chaining. A list was used to implement the hash table because I believe it is the most appropriate python data structure to use for open addressing. Although in this assignment the keys are integers, it is possible that in some situation, the keys may be strings or maybe we need to insert a key-value pair in the form of a tuple or list. In these situations, a list would be useful because it does not restrict the data type of its elements. Thus, the list data structure was chosen because of its flexibility. An alternative to list would be a NumPy array.

I chose to implement the hash functions and collision handling methods all as class methods of the hash table class because it allows the different methods to share class attributes which allows the methods to work more coherently with each other. For example, the hash table class has a class attribute "bucket" that stores the bucket size of the table. This information can be shared among all methods in the class so that I don't have to enter the bucket size as argument every time I use one of the methods. Each object method will automatically adjust its procedure according to the bucket size of the hash table object. This approach also allows me to better keep track of the relevant statistics such as the number of collisions and comparisons by allowing these statistics to be shared and updated by all methods in the class. Logically speaking, it makes sense to include these method into the hash table class since the only time I use them is when I am using the hash table object. The stack object was chosen to keep track of free space in the table because stack is one of the simplest and easiest data structure to implement. Thus, it was chosen for its simplicity.

The main function takes care of the input/output stream by first importing all records into a python list. Since the number of records in a file is usually under 120, reading all records into a list will not take up that much memory. This list of records are then inserted into a hash table using an auxiliary function "hash_scheme". The insertion was repeated 11 times to hash the records from the same input file into 11 separate hash tables. Each table corresponds to 1 of the 11 schemes specified in the lab handout. All of the 11 hash tables are then printed to an output file that is specified by the user. In addition to the "hash_scheme" function, a printList and a printHash function were also written as auxiliary functions. The printList function was used to print out the original input records to the output file. It is also used to print out the records that could not be inserted into the hash table. The printHash prints the hash tables from the 11 schemes to the designated output file along with the relevant statistics. Both of the print functions were incorporated into the hash_scheme function. Since the hash tables are always

printed after the records were inserted, it makes sense to incorporate the print functions into the hash_scheme function. All 3 auxiliary functions were saved in a file that is separate from where the main function is saved. The purpose of this is to make sure that the main function is not too crowded and is clean and easy to understand to the reader.

Hashing Functions

Division and multiplication methods were used in this assignment. Division uses the following formula:

$$h(k) = k \bmod m$$

The division method is one of the most popular and effective hashing methods. It is also very easy to implement and is quite fast since it only requires a single division operation. However, the ramification of using division is that the value of modulo, m, matters. For example, m should never be a power of 2. Because if it is, then dividing a key by m=2^p will just result in a hash value that is the p lowest bits of the key. Thus, we would be cutting off parts of the data in the key and compute hash values that may be biased. So unless if we are sure that all keys differ in the p lowest bits, it is better to use a m value that uses all of the bits. Further, the value of the records to be inserted also matters. If all records end in 0 then dividing the records by a m value of 10 would result in the same hash value for each record.

Multiplication is another popular hashing method. It uses the following formula:

$$h(k) = floor(m \times (kA \bmod 1))$$

where $0 < A < 1$

The multiplication method is similar to a random number generator. One advantage that it has over division is that the m value does not matter. It could technically work for any A but setting A as $(\sqrt{5} - 1)/2$ works pretty well generally. However, because it involves more constant time operations than division, it is usually slower than division.

Collision Handling Methods

Linear probing, quadratic probing and chaining were used in this assignment. It is easy to implement linear probing and, as we will see in later sections of this analysis, it only leads to a moderate amount of collisions when the table is not very full. However, it usually causes primary clustering which is a condition in which long runs of occupied slots build up which leads to increased search time and number of collisions as the table gets full. Long runs of occupied slots tend to get longer as the table gets full.

Quadratic probing avoids primary collision by jumping an increased amount of slot the more collision it encounters. However, one drawback of quadratic probing is that not all locations in the table can be generated by each key. For example, if the table size is 10 and a key with value 7 is probed 10 times using division modulo 10 as the auxiliary hash function in the probing formula, quadratic probing will not generate location 1,4,6 and 9. As a result, not all of the records are guaranteed to be inserted even if there are enough slots in the table for all. As we will see in later sections, this phenomenon is also observed in our results.

When using probing, uniform hashing is preferred. That is, given a hash table with size m, the probe sequence of each key should be equally likely to be any of the m! permutations of (0,1,..... m-1) to minimize collisions. However, in both linear and quadratic probing, the initial probe determines the entire probe sequence and so there are only m probing sequences for both linear and quadratic probing.

In this assignment, the linked list of chaining is stored inside the table. A ramification of doing chaining this way is that, in the extreme case, several chains could be merged into one and we could end up with an extremely long chain. In the worst case, searching for an item in the linked list could take O(n) time for a table with n stored items.

Input files

12 input files containing random integers were generated for this analysis. The number of records in each file range from 10 to 120. For each input file, data from all 11 schemes were computed. The collision statistics shown in this analysis was calculated from summing the primary and secondary collisions of each insertion. For chaining, each node in a linked list is counted as a comparison when a record is inserted into a chain. During probing, each probe in the probe sequence is counted as a comparison including the probe in which the record is inserted.

A comparison of the number of collisions between the different schemes

Number of collisions

number_of_records	scheme1	scheme2	scheme3	scheme4	scheme5	scheme6	scheme7	scheme8	scheme9	scheme10	scheme11
10	2	2	2	2	2	1	0	0	0	0	0
20	1	1	1	4	4	4	0	0	0	0	0
30	4	8	3	5	5	4	1	1	4	4	4
40	8	11	9	6	6	5	0	0	3	3	3
50	17	19	12	15	15	12	7	7	7	7	6
60	17	16	12	46	40	14	9	10	39	36	20
70	41	33	18	40	33	20	7	7	49	42	25
80	117	80	33	100	61	25	21	26	122	84	29
90	60	76	30	172	108	36	45	37	154	104	40
100	185	127	38	238	133	42	35	37	153	118	44
110	218	156	43	397	157	49	170	81	351	211	51
120	581	266	56	811	515	59	311	184	660	443	65

The above table shows the number of collisions for each scheme. Graphs were also made for better visualization (see the supplemental materials section). In general, all schemes performed similarly in terms of the number of collisions when the number of records inserted was low (between 10-50). Among the schemes with bucket size 1, the ones that used multiplication method (scheme 9,10 and 11) tend to do slightly better than the division schemes when the number of records inserted was low. The differences in the number of collisions between each scheme became apparent when the number of records was greater than 50. When the number of records to be inserted was high, the multiplication schemes had more collisions than the division schemes in general. In particular, scheme 9,10 and 11 had more collisions than their corresponding division schemes 1, 2 and 3. This suggests that division may be a slightly better hashing method to use than multiplication. Scheme 4, 5, 6, 7 and 8 also used division for hashing. However, scheme 4, 5 and 6 used 113 as divisor instead of 120. The implication of that will be discussed later in this section. Scheme 7 and 8 will be discussed in the next section.

The effect of the value of the divisor used in the division method can be observed when we compare the differences between scheme 1,2 and 3 and scheme 4, 5 and 6. Scheme 4, 5 and 6 used 113 as their divisor instead of 120. When we use modulo 113, we are effectively limiting the range of primary hash values to

0-112. This is similar to shrinking the table size. As mentioned in the lectures, when we shrink the table size, we increase the likelihood of primary collisions. When we increase the number of records to be inserted, scheme 4, 5 and 6 will have more collisions than scheme 1, 2 and 3 which used modulo 120 because the number of slots that can be primarily hashed to in the table is less in 4, 5 and 6.

In general, scheme 1, 4 and 9 had the most collisions out of all schemes. All of these 3 schemes used linear probing as their collision handling method. As mentioned before, linear probing tend to form primary clusters and long clusters tend to get longer as the number of stored records increase. As the clusters build up in the table, we would expect more collisions to happen when we insert a record, especially when a primary collision happens and we have to go down the clump to find the next available slot. Thus, it is not surprising that the number of collisions increased the most in linear probing out of all schemes as the number of records to be inserted increased. Scheme 7 also used linear probing but since it had a bucket size of 3, the number of collisions were drastically reduced because a collision only happens when a record is inserted into a slot that already stores 3 records. However, as we will see later in this section, this advantage of having a bigger bucket size is offsetted by the higher number of memory accession required. The combination of linear probing and using 113 as the divisor may be the reason why scheme 4 had the highest number of collisions.

In general, the schemes that used quadratic probing as their collision handling method tend to have less collisions than the schemes that used linear probing. As mentioned earlier, quadratic probing avoids primary clustering by "jumping" in increasing amount in the table as it encounters more collisions. Since primary clustering is avoided, the number of collisions in quadratic probing is expected to be less than the number of collisions from linear probing. This is, in fact, what was observed from the results. However, quadratic probing does not always guarantee that all locations will be filled. When there were 120 records to be inserted into the table, none of the quadratic probing schemes were able to insert all of the records even though there were 120 slots in the table. Nevertheless, the number of records that were unable to be inserted were little compared to the number of records that were inserted. Thus, this is not a major problem.

As expected, the schemes that used chaining (i.e. 3, 6 and 11) had considerably less collisions compared to the other 2 collision handling methods. Since the records in the inputs were randomly generated, a scenario in which all chains merge into one is unlikely to happen. This was reflected in the small difference between the number of comparisons and the number of inserted records in the chaining schemes (data not shown here). The number of comparisons were close to the number of inserted records for all chaining schemes regardless of the number of records in the input files. This suggests that the length of the chains in each chaining scheme was short. Note that in the chaining schemes, the number of comparisons reflects the number of nodes that were traversed in a chain. As a comparison, I have included an input file that consists of 121 records that are the same value. The results from that input showed that the number of comparisons were much greater than the number of records inserted in the chaining schemes. This is what one would expect when all records hash to the same slot and are in the same chain since for each insertion into the chain, the program has to go through O(n) nodes for n equals the number of records stored.

Comparing the number of comparisons from each scheme

Number of comparisons

number_of_records	scheme1	scheme2	scheme3	scheme4	scheme5	scheme6	scheme7	scheme8	scheme9	scheme10	scheme11
10	12	12	10	12	12	10	13	13	10	10	10
20	21	21	20	24	24	20	23	23	20	20	20

30	34	38	30	35	35	30	44	44	34	34	31
40	48	51	40	46	46	40	49	49	43	43	40
50	67	69	51	65	65	52	101	103	57	57	50
60	77	76	61	106	100	63	124	128	99	96	65
70	111	103	72	110	103	72	140	141	119	112	79
80	197	160	96	180	141	86	210	226	202	164	88
90	150	166	102	262	198	113	303	279	244	194	110
100	285	227	109	338	233	109	291	297	253	218	110
110	328	266	125	507	267	130	722	454	461	321	145
120	701	379	170	931	632	150	1173	789	780	561	152

The above table shows the number of comparisons for each scheme. Graphs were also made for better visualization (see the supplemental materials section). In general, the number of comparisons in each scheme was consistent with the number of collisions. Schemes that were high in the number of collisions were also high in the number of comparisons and vice versa. Just like in collisions, all schemes seemed to perform similarly when the number of records to be inserted were no more than 50 except for scheme 7 and 8. As mentioned earlier, schemes with a bucket size of 3 had less collisions. However, their number of comparisons were the highest among all schemes. This suggests that scheme 7 and 8 require a lot more memory access compared to the schemes that used a bucket size of 1. Thus, the advantage of having low collisions was offsetted by their high number of comparisons. In other words, scheme 7 and 8 may be appealing in theory due to their low collision rate, but in reality, the number shows that they require a much higher amount of memory access and therefore are not any faster than the other schemes.

As mentioned earlier, all of the schemes seemed to have performed similarly both in terms of collisions and comparisons when the number of records were low (i.e. 50 or under). However, an observed problem that was common among all schemes was that the number of collisions increased drastically when the number of records to be inserted was high relative to the table size. One way to solve this problem and keep the efficiency of insertion high is by re-sizing the hash table when the number of records stored reaches a certain threshold. By doing so, we can ensure that hashing can be performed at a low runtime time using any of the methods from the schemes mentioned above.

It was mentioned in the lecture that a quick and easy way to implement hashing is to use the division method and linear probing as the collision handling method. The data from this analysis suggests that this may be true, assuming if the number of records stored is low relative to the table size. As the results have shown, scheme 1 and 4 performed similarly to the other schemes in terms of collisions and comparisons when the number of records was low. Thus, given how easy it is to implement the division method and linear probing, I believe using these 2 methods is an easy and effective way to implement hashing.

A comparison of theoretical and observed efficiency

As the text book had already explained in chapter 11, inserting an element into hash table that uses linear or quadratic probing as the collision handling method with load factor α requires at most $1/(1-\alpha)$ probes on average, assuming uniform hashing. Since in open addressing $\alpha \leq 1$, the runtime for insertion is constant (i.e. O(1)). Insertion into a table that uses chaining should also take constant time (i.e. O(1)).

Although the number of collisions and comparisons observed from this analysis showed that the runtime of hashing may be slower than in theory, insertion using the different collision handling methods should still take constant time. In the case of probing, a comparison between the number of records inserted and the number of collisions from the results showed that, in the worst case, the number of collisions was

larger than the number of inserted records by a constant factor and it was not exponentially larger than the number of records inserted. This reflects the fact that, on average, each inserted record only encountered some constant number of collisions. For example, when 120 records were inserted into a table with 120 slots, in the linear probing schemes the number of collisions was about 5-7 times the number of records inserted (i.e. 120). In the quadratic probing schemes, the number of collisions was about 3-4 times the number of records inserted. Therefore, the runtime should only increase at some constant level. Thus, even though each insertion may have encountered some number of collisions, the insertion time should still be constant in reality.

As mentioned in the previous section, the expected length of the chains in the chaining schemes should be quite short based on the comparison between the number of comparisons and the number of inserted records. Thus, since the number of comparisons was quite close to the number of inserted records (no more than x1.5 the number of inserted records) across all chaining schemes, the time it takes to insert into the end of a chain should still be constant since the runtime of insertion only increased at some constant level due to the chain. Note again that in chaining, the number of comparisons reflects the number of nodes that were traversed in a chain when inserting into the chain.

Ramification of deleting a record from the table

In the case of chaining, the chains were implemented as singly linked lists. Since the chains were implemented as singly linked lists, it does not support deletion very well. To delete a record from the chain, the program would need to first traverse through the chain to find the record so that it could update the record's predecessor to point to the record's successor. Thus, the time it takes to delete a record in the chain is proportional to the length of the linked list.

Deletion would become more of a problem in linear and quadratic probing because when we delete a record from the table, we cannot simply mark the slot as empty. If we did, then we might not be able to retrieve any records during whose insertion we had probed the deleted slot and found it occupied. This is because in open addressing, the algorithm for searching a key probes the same sequence of slots that the insertion algorithm examined when that key was inserted. Therefore, when searching for a record in a hash table, the search terminates when it encounters an empty slot since the record would have been inserted there and no later in the probe sequence. We could solve this problem by marking the deleted slot with a special value such as "deleted" or "tombstone" so that when the search algorithm encounters this special value, it knows that it should just go to the next slot in the probe sequence instead of terminating. We then have to modify our insert function to treat slots with these special values as empty so that records could be inserted. However, if we decide to use this approach, then the search time would no longer be dependent on the load factor of the table.

Enhancement

Several things were done as enhancement to this program. Please see below:

1) statistics on the number of records inserted were provided in the output files along with the load factor.
2) An extensive amount of input files were generated for data collection and analysis purposes. Other types of inputs were also generated to test error handling and the extreme cases
3) 8 different graphs were made to better visualize and analyze the data.
4) The output file were printed in clear, well organized format to help reader understand the results.

What I have learned from this assignment

The most important thing I have learned from this lab is the relationship between load factor and the number of collisions when inserting into an open address table. In an open address table the load factor cannot be greater than 1 (except in the case of bucket size >1). The load factor is therefore an indication of how full the table is because it tells you the number of records stored relative to the total number of

slots available. Based on the results, it is obvious that the number of collisions remains low when the number of records to be inserted is few compared to the number of slots in the table. However, as the table becomes full, the number of collisions and comparisons drastically increases. This teaches me that if I want the search and insert functions to work efficiently in a hash table, I should always make sure that the table is not full. Based on the results from this analysis, the table should not be more than ½ full.

What might I do differently next time

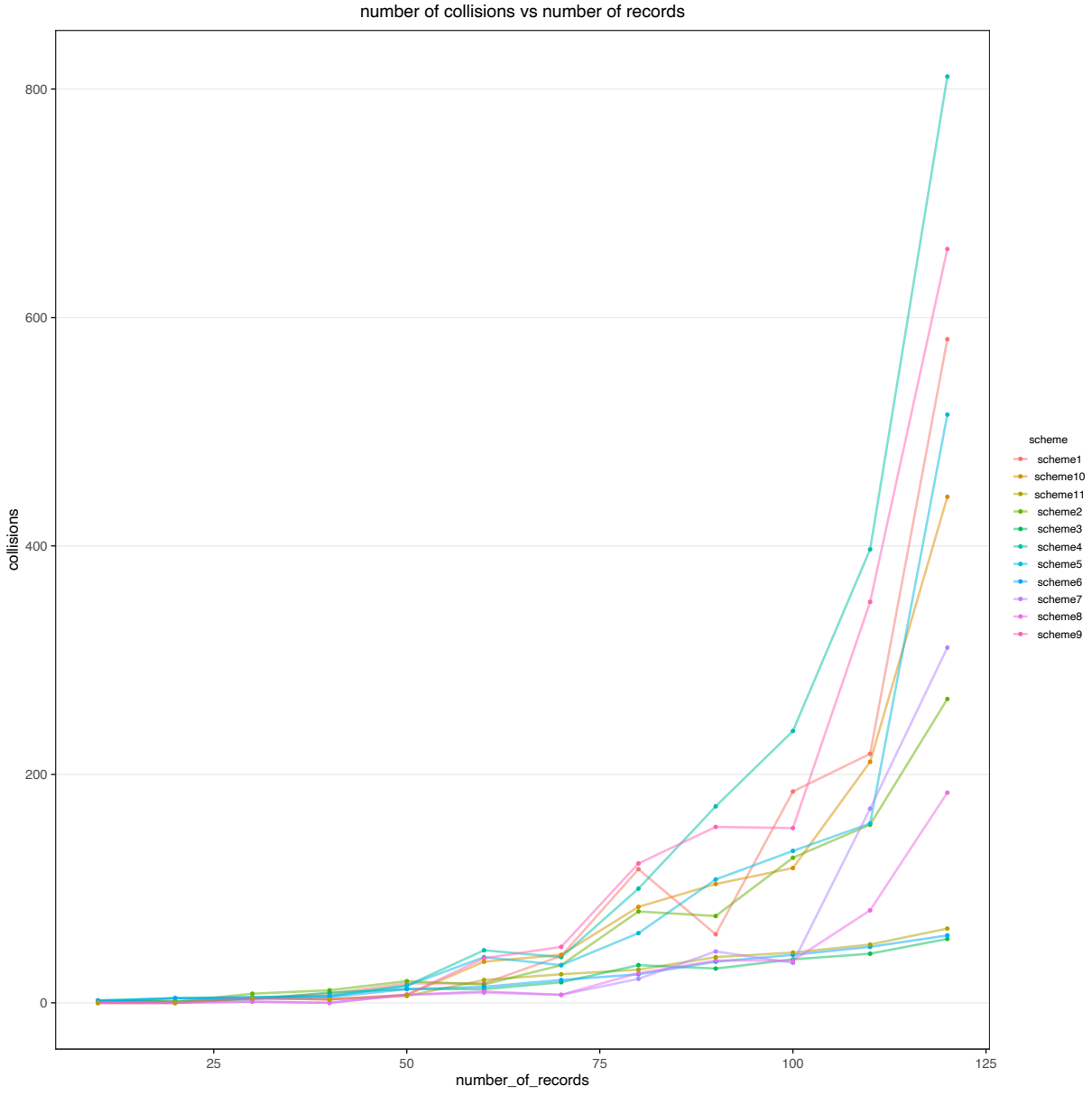
I may try to implement my chain using a doubly linked list instead of a singly linked list. In this assignment, since we are not removing any records from the table, it does not make a big difference whether we use a singly linked list or a doubly linked list. However, from a more practical perspective, it may be better to use a doubly linked list since it allows items in the list to be deleted faster. In reality, depending on how we use a hash table, it is possible that deletion from the table may happen frequently.

Application in bioinformatics

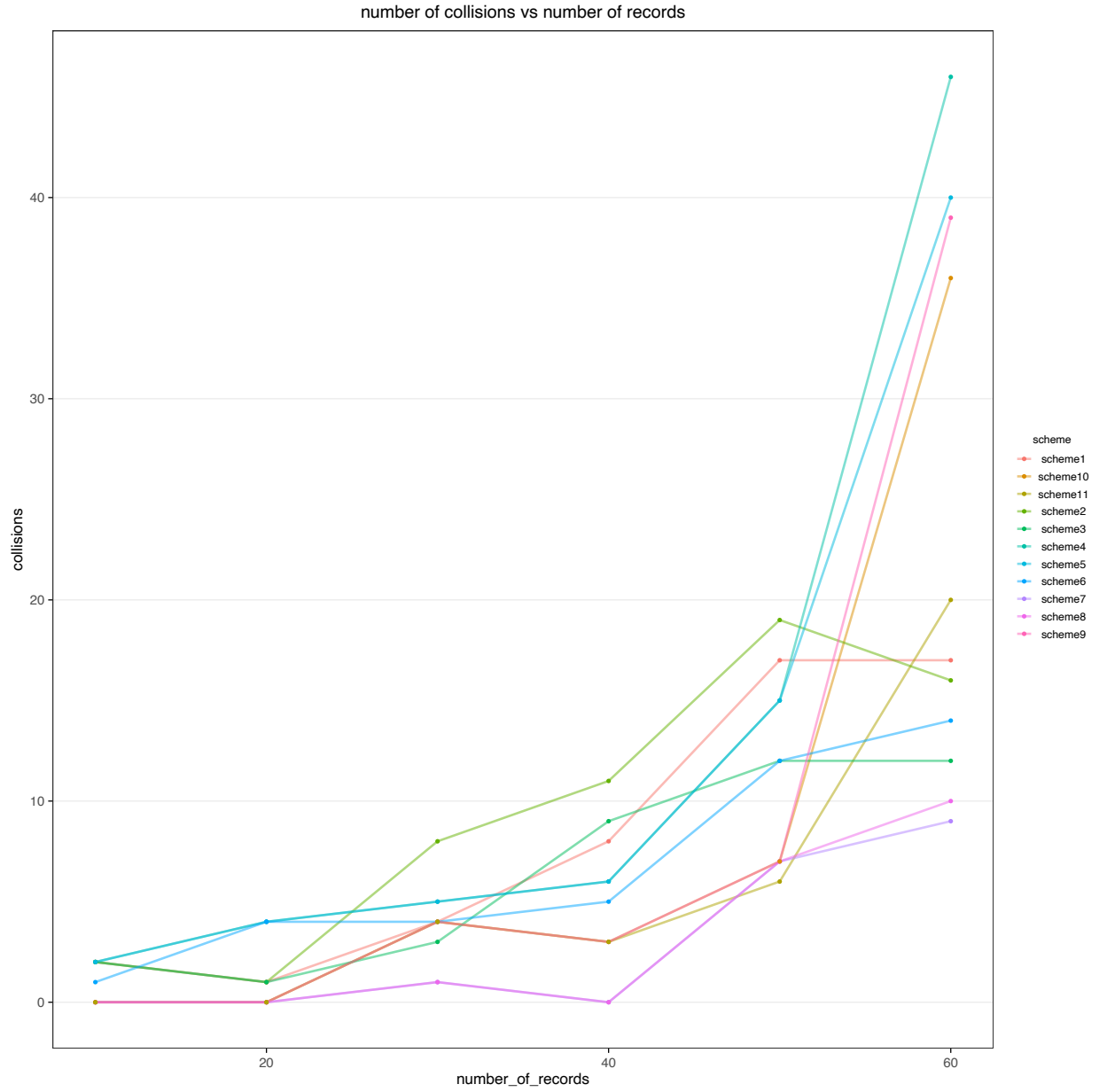
Hash table is widely used within the bioinformatics community. One of its applications is in digital normalization of metagenomic data. Metagenomics is the study of recovering all of the microorganisms directly from an environmental sample without culturing. This discipline relies heavily on high throughput nucleotide sequencing techniques. Often, when we are trying to sequence all of the microorganisms from an environmental sample, some of the microorganisms are over-covered while others may be under covered due to sampling bias. Thus, in order to sufficiently cover all of the microorganisms, deep sequencing of the environmental sample is required. This leads to an unnecessarily high amount of reads in some of the organisms which takes up a lot of the memory and slows down the processing speed of genome assembly. Further, sequencing reads often contain errors. A high number of reads with errors makes genome assembly more difficult. In theory, much of the high coverage data are redundant and can be removed. But determining which reads to remove requires an estimation of its coverage. Digital normalization identifies and removes redundant reads by looking at the median k-mer abundance within individual reads. This stems from the idea that the more times a region is sequenced, the higher the abundance of k-mers from that region would be. Thus, reads from a high coverage region should have a higher median abundance of k-mers. We retain only those reads that has an estimated coverage below the threshold that we set for the dataset. Using this method, digital normalization removes redundant reads and normalizes average coverage of the dataset to a specified value. By doing so, the size of the dataset is greatly reduced, thereby increasing the processing speed and reducing the memory requirement. Digital normalization uses the khmer package which relies on count-min sketch as its underlying data structures to count k-mers. Essentially, khmer uses a set of hash tables, each of a different size. These hash tables are used to store the frequency of specific k-mers.

Supplemental materials

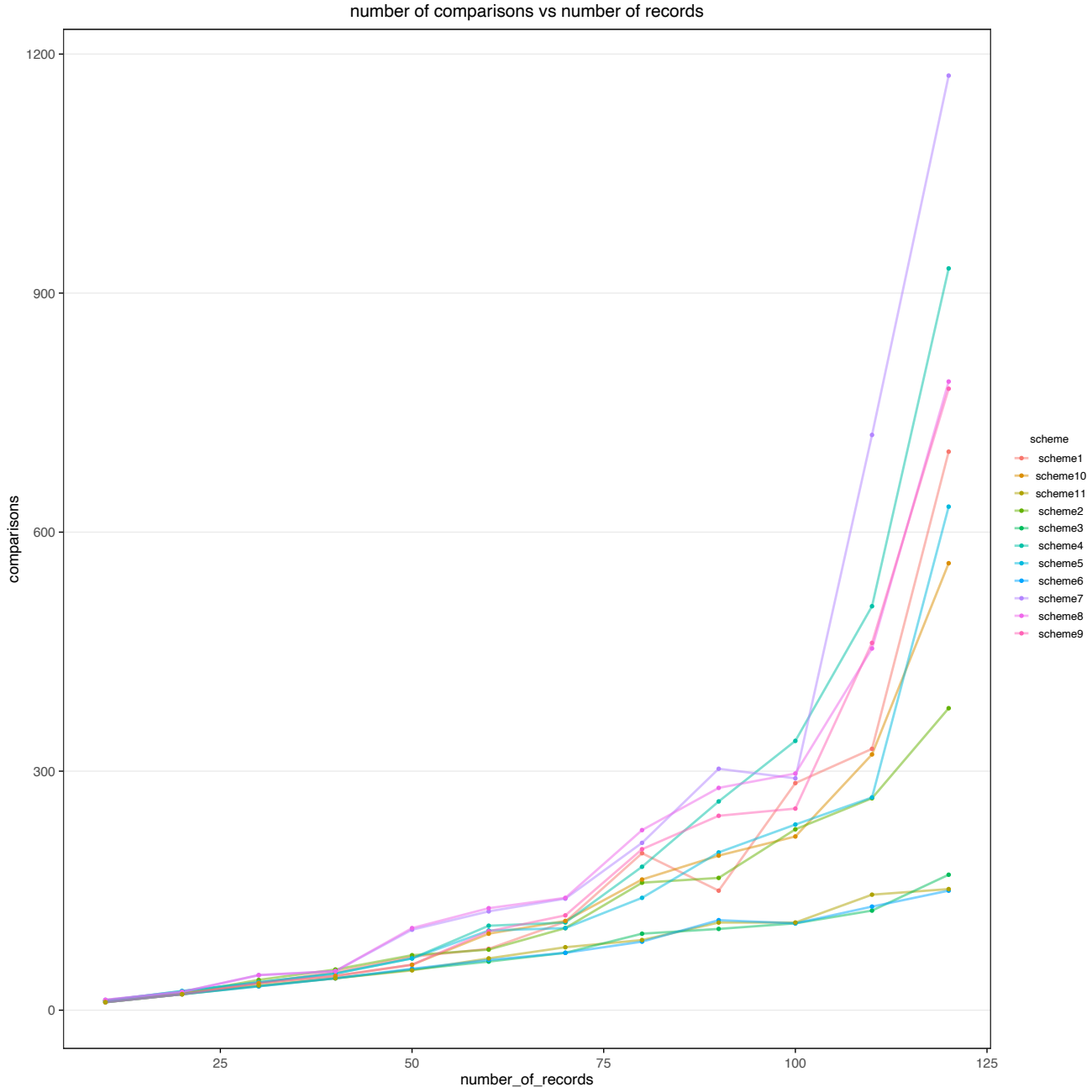
Number of collisions for each scheme with records from 10 -120



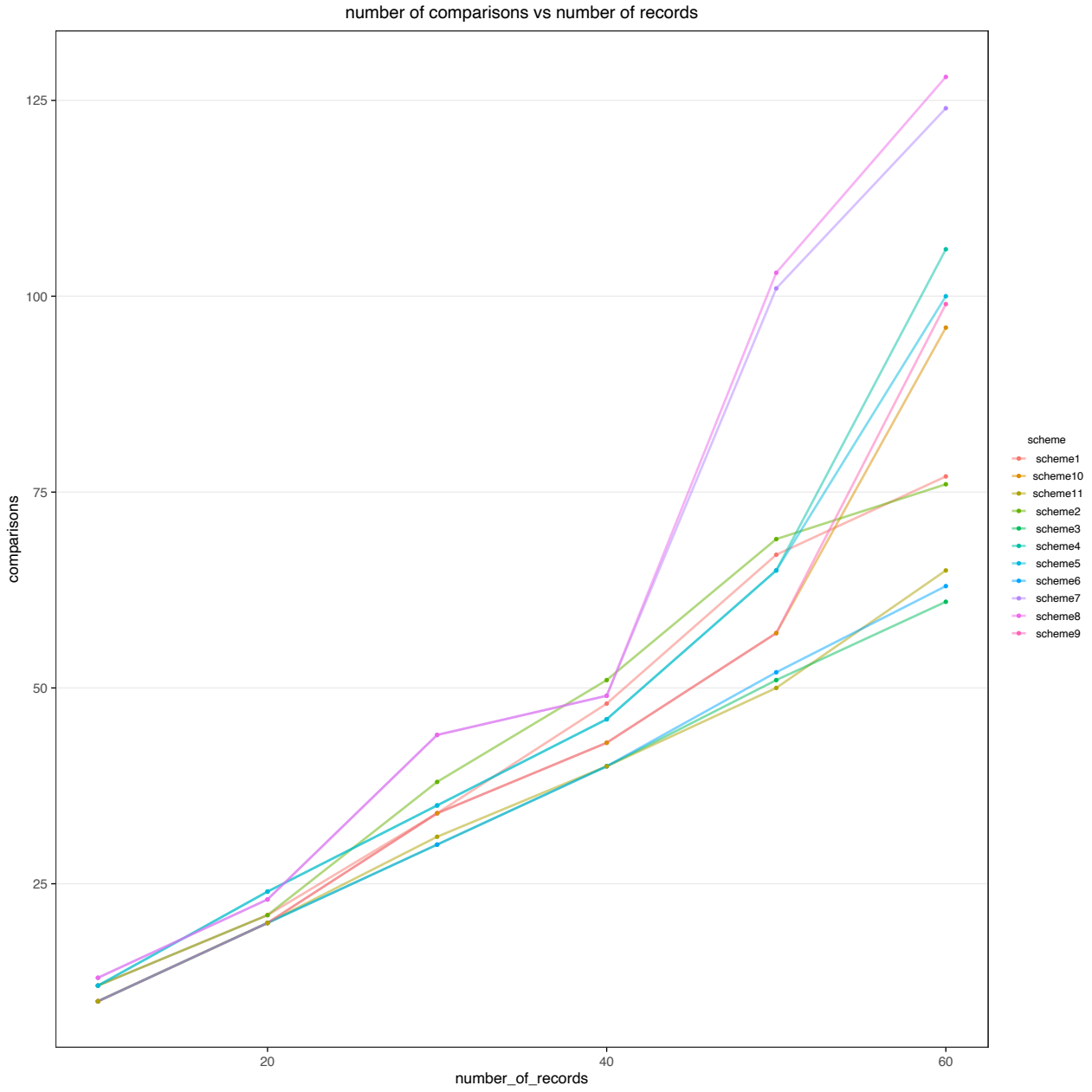
Number of collisions for each scheme with records from 10 - 60



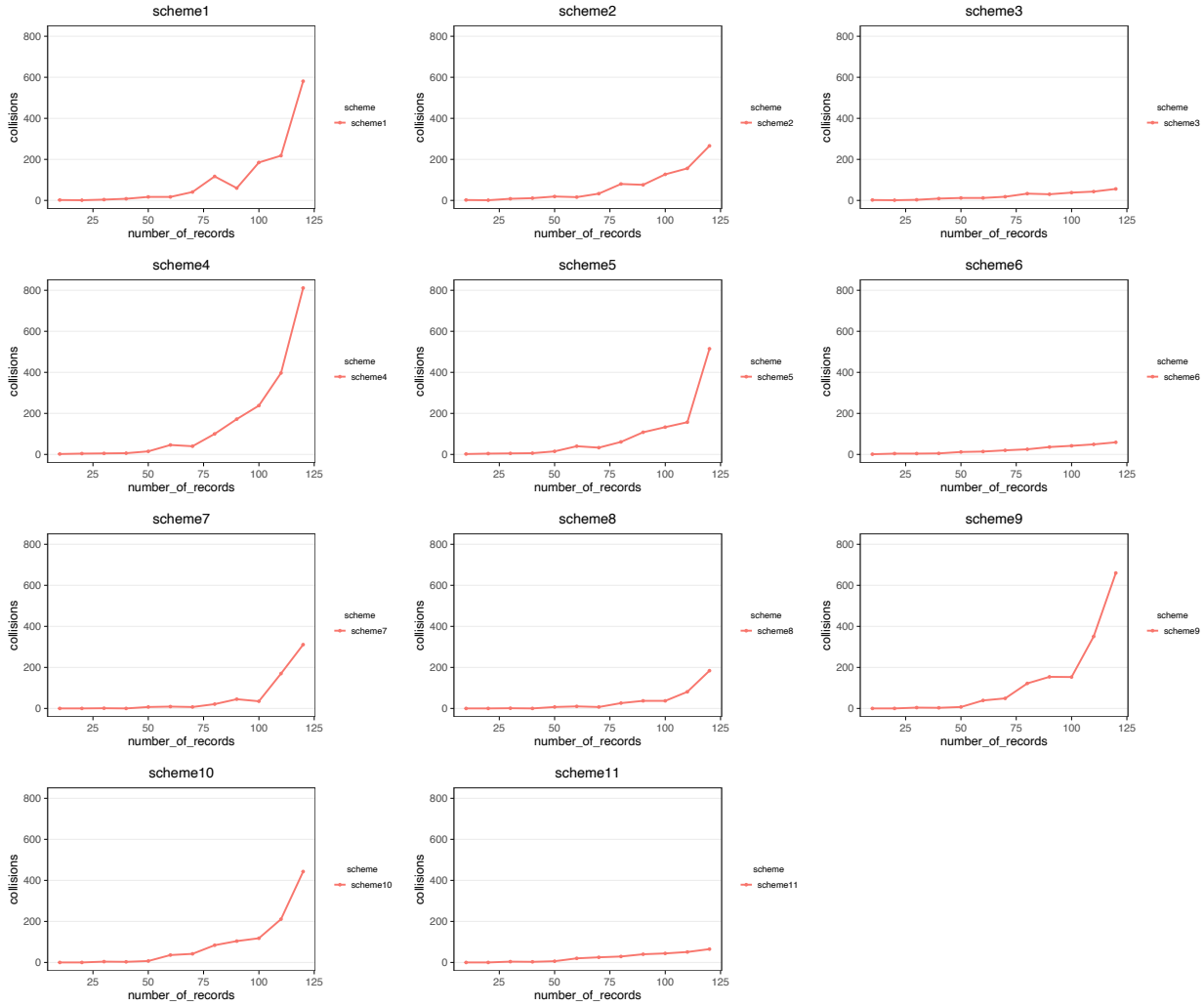
Number of comparisons for each scheme with records from 10 - 120



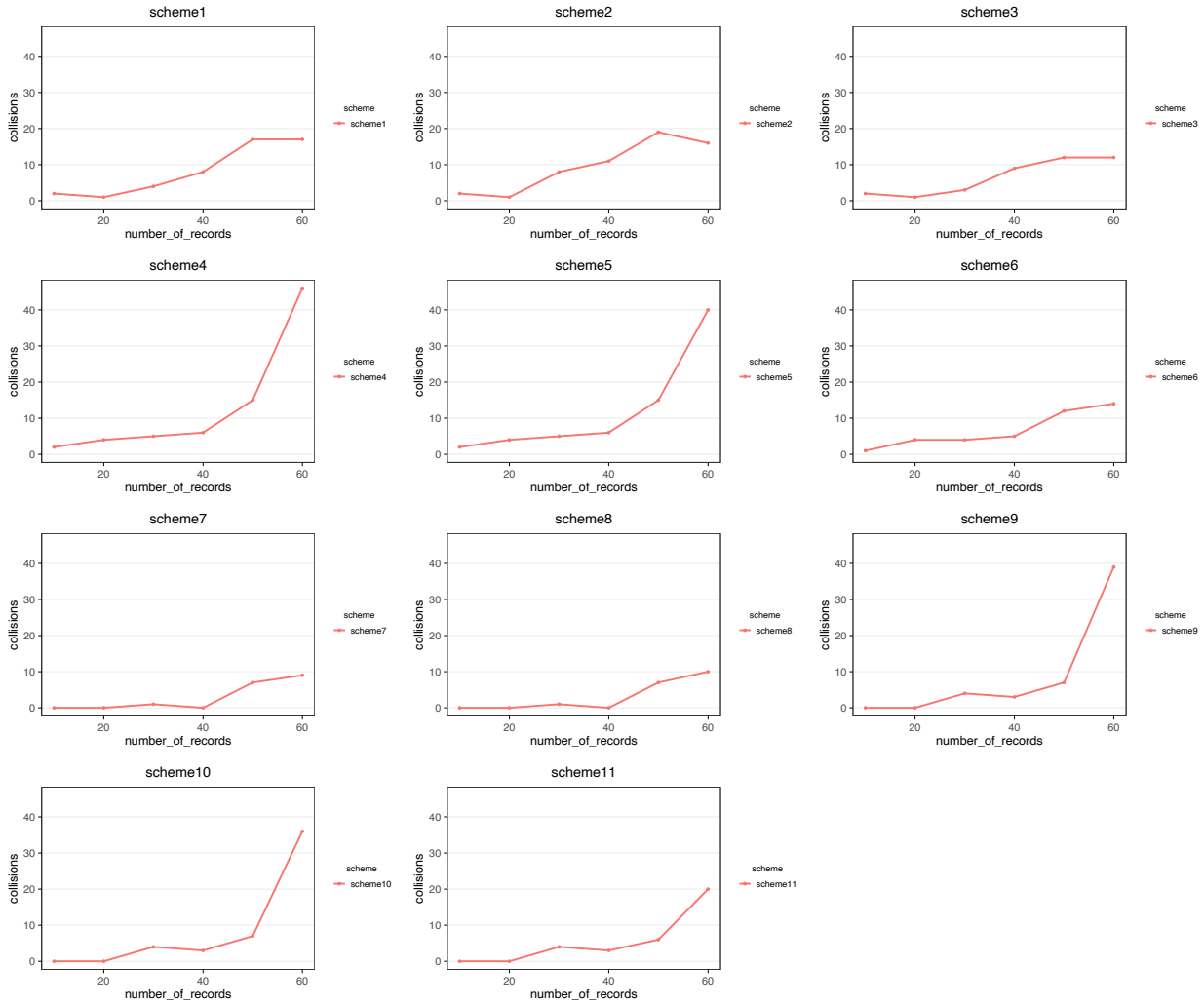
Number of comparisons for each scheme with records from 10 - 60



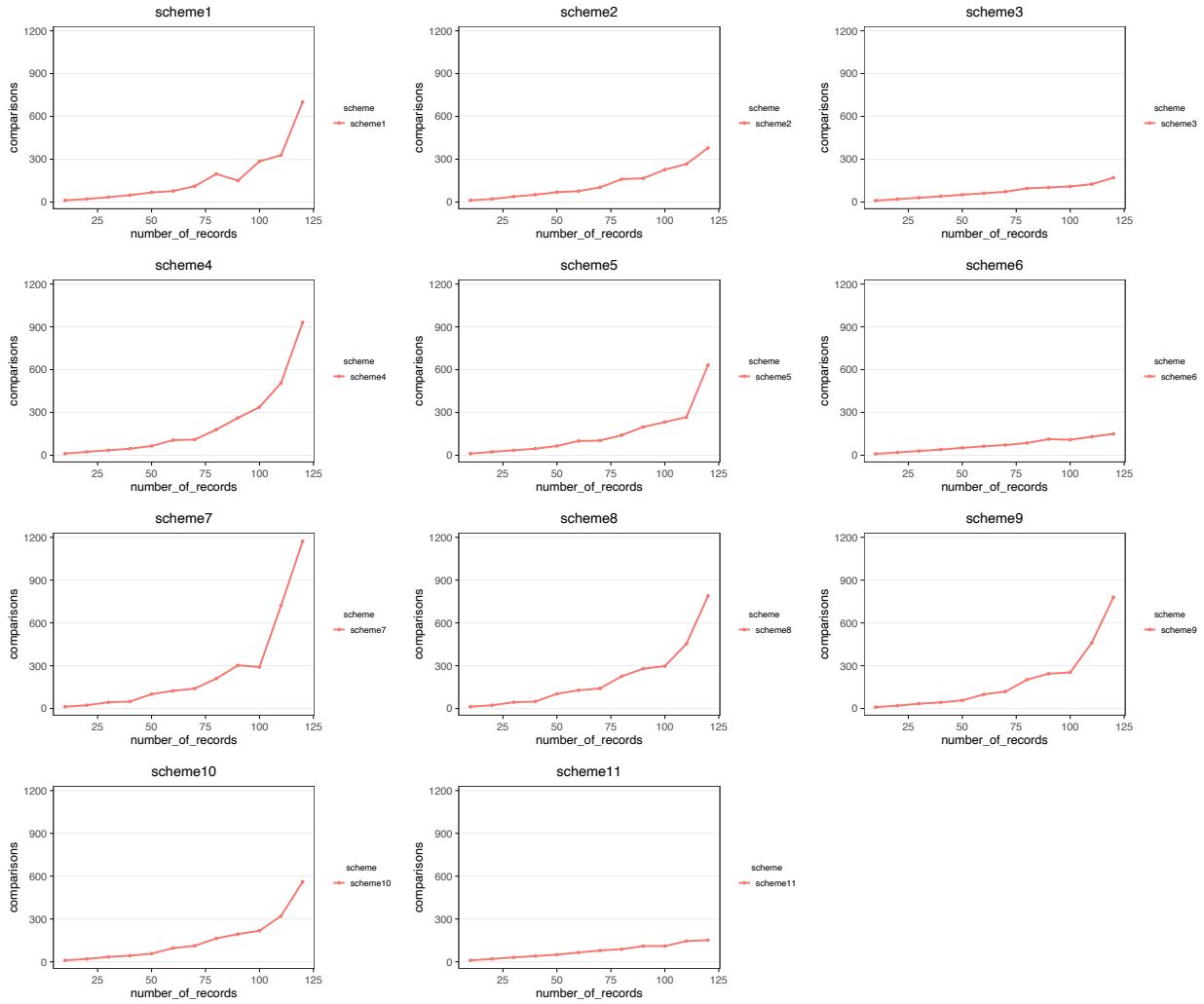
Individual plots for each scheme showing the number of collisions (number of records from 10-120)



Individual plots for each scheme showing the number of collisions (number of records from 10 - 60)



Individual plots for each scheme showing the number of comparisons (number of records from 10 - 120)



Individual plots showing the number of comparisons for each scheme (number of records from 10 - 60)