

A Performance Test of Iceberg Cubing Algorithms

Xiaolei Li

xli10@uiuc.edu

July 25, 2003

Abstract

In data mining, the computation of a data cube is one of the most fundamental steps. It is often unrealistic to compute the entire cube when faced with high dimensionality and high cardinality. The iceberg cube is an effective method to compute only aggregate group-bys above a certain threshold. Currently, several methods exist for computing iceberg cubes. They use efficient methods that exploit various properties of the data cube and measure, such as *anti-monotonicity*. In this paper, we will study four methods and thoroughly compare their performances vs. each other under various types of data.

1 Introduction

The data cube is an integral part of the data warehouse. Many decision support systems precompute it to improve response time. A complete materialization of the cuboids in a decent-sized system would require a tremendous amount of storage space and running time. Therefore, it would make much more sense to only partially compute the cuboids. One proposed solution is the Iceberg-CUBE. Many low level cuboids offer only trivial aggregate values and thus do not need to be computed. Doing this would save both time and space.

Three primary algorithms have been proposed for the data cube and Iceberg-CUBE problems. The first and oldest, Multi-Way Array (ArrayCube) [4], computes the complete data cube. But it puts the data into smaller multidimensional chunks and computes them in an order that minimizes memory usage. The second method, BUC [1], uses the Iceberg-CUBE concept and starts from the bottom of the data cube lattice and slowly works towards the least aggregated group-bys. Along the lattice, it checks at every step to make sure the current partition meets a threshold. If it does not, the algorithm skips the partition since any ancestors of it would be smaller. The last algorithm, H-Cubing [3], uses a hyper-tree data structure, called H-tree, along with quant-info at each node. Two variants of this algorithm perform the algorithm using either a bottom-up or top-down approach.

In this paper, we will study all four of these methods and thoroughly compare their performances vs. each other under various types of data. In Section 2, we will give a quick overview of the methods. In Section 3, we present all our performance data. In Section 4, we will discuss some future work, and we will conclude in Section 5.

2 Overview of Algorithms

The *cube* operator was first proposed by Gray, et al. [2] in 1995. In 1997, Y. Zhao, et al. [4] proposed the multiway algorithm for computing the cube using an array-based method. In

1999, K. Beyer and R. Ramakrishnan [1] proposed BUC, a bottom-up computation of the iceberg. And in 2001, J. Han, et al. [3] proposed their cubing algorithm using the H-Tree. The bottom-up version of H-Cubing was proposed first and the top-down version came later. We shall give a brief overview of the four algorithms below.

2.1 Multi-Way Array

The first method is an array-based cubing algorithm. In it, the base table is loaded into arrays and the cube is computed from the resulting array. By doing everything with arrays, no tuple comparisons are needed. Only the array index is needed. In order to save memory usage, the authors divide the arrays into *chunks*. The idea is to divide a single large n-dimensional array into a much smaller n-dimensional array. In this manner, the algorithm uses a single pass over the entire data and uses the chunked arrays to compute the data cube. It is not necessary to keep all the chunks in memory; only parts of the group-by arrays are needed. In addition, multiple cuboids can be computed simultaneously in the pass. In case of sparse arrays, they are compressed in order to save space. A hash table is then built on top of the compressed arrays to facilitate fast lookups.

The exact computation of the cube will roughly follow the set of steps below. First, the dimensions are ordered from least to greatest in terms of cardinality. This will minimize the amount of memory required during the computation. Then starting at the base cuboid, a single chunk will be processed. Its results will be passed to the immediate lower level cuboids and so forth. Once all computations of that chunk has completed, the base cuboid will read the next chunk from the file and repeat. One thing to notice is that the multi-way array does not consider the iceberg condition until the very end. There is no pruning involved.

2.2 Bottom-Up Computation

Bottom-Up Computation (BUC) employs a bottom-up method, as the name implies. It starts at the apex cuboid and moves upward to the base cuboid. The algorithm starts by reading the first dimension and partitioning it based on its distinct values. This partition is facilitated by CountingSort, an $O(n)$ sorting algorithm in sparse data. Then for each partition in the first dimension, the algorithm recursively computes the remaining dimensions.

During partitioning, each partition's size is compared to the minimum support. If the count does not satisfy the minimum support, the recursion will stop. This is because further partitioning into more dimensions will return less aggregated group-bys, whose count cannot support minimum support. This pruning technique can save many unnecessary computations.

2.3 H-Cubing

H-Cubing uses a hyper-tree structure, called H-tree, to facilitate the cube computation. Each level in the tree represents a dimension in the base cuboid. Two nodes have a descendent-ancestor relationship if they appear in the same tuple in the base cuboid. Nodes in the same level of tree that hold the same value, i.e., cousins in the tree that represent the same value for that dimension, are linked together via a side-link. Each node in the H-tree holds a *quant info* that basically counts the number of times that particular value occur in the

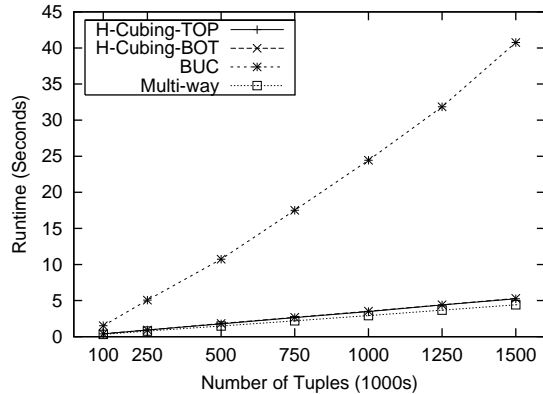


Figure 1: **Tuple Size** $\mathcal{D} = 5$, $\mathcal{C} = 5$, $\mathcal{S} = 0$, $\mathcal{M} = 1$

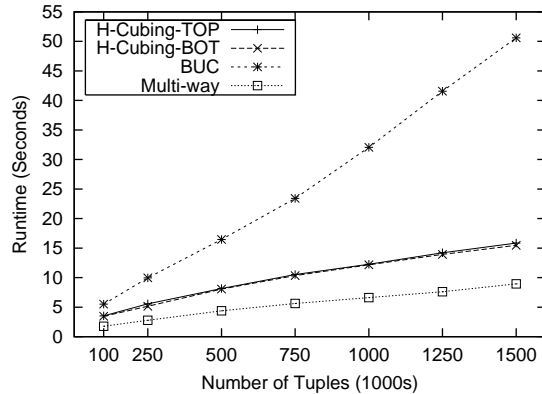


Figure 2: **Tuple Size** $\mathcal{D} = 5$, $\mathcal{C} = 15$, $\mathcal{S} = 0$, $\mathcal{M} = 1$

dimension. In addition to the H-tree, a Header Table coexists that records the count of all distinct values in all dimensions and provides a link to the first node of a particular in the H-tree.

Using this data structure, there are two methods. The first is a bottom-up approach and the other is a top-down approach. In both methods, you start at a particular level of the H-tree, i.e. a particular dimension, and examine group-bys that include that particular level and levels that are above it in the H-Tree. The counting in the aggregations are facilitated by the Header Table constructed in the initial pass of the data and Header Tables local to the current group-by. While performing the aggregation, if a particular notices that its count is below the minimum support level, it can skip itself and move on to the next node via the side-link. The difference between the two approaches, H-Cubing BOT and H-Cubing TOP, is that H-Cubing TOP starts the process at the top of the H-tree while the other starts at the bottom.

3 Performance Tests

In this section, we will examine all the performance tests we executed. All four algorithms were coded using C++ on a AMD Athlon 1.4GHz system with 512MB of RAM. The system ran Linux with a 2.4.18 kernel and g++ 2.95.3. The times recorded include both the computation time and the I/O time. All the tests used data such that the algorithms could fit in main memory.

For the rest of this section, \mathcal{D} will denote the number of dimensions, \mathcal{C} will denote the cardinality of each dimension, \mathcal{T} will denote the number of tuples in the base cuboid, \mathcal{M} will denote the minimum support level, and \mathcal{S} will denote the skew or zipf of the data. When \mathcal{S} equals 0.0, the data is uniform; as \mathcal{S} increases, the data is more skewed. \mathcal{S} is applied to all dimensions in a particular data set.

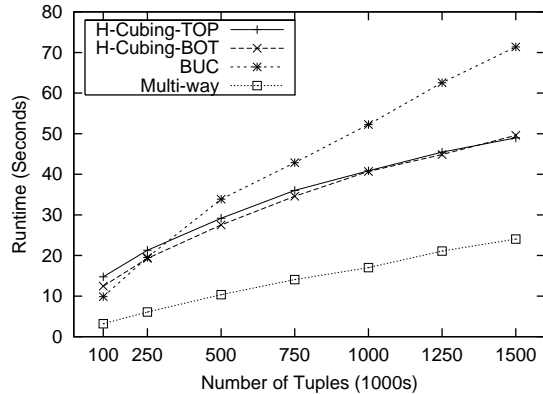


Figure 3: **Tuple Size** $\mathcal{D} = 5$, $\mathcal{C} = 25$, $\mathcal{S} = 0$, $\mathcal{M} = 1$

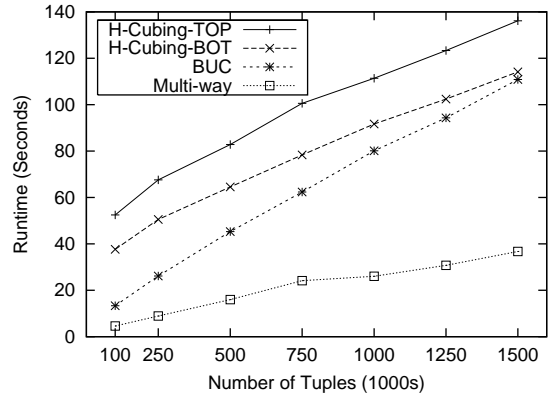


Figure 4: **Tuple Size** $\mathcal{D} = 5$, $\mathcal{C} = 35$, $\mathcal{S} = 0$, $\mathcal{M} = 1$

3.1 Tuple Size and Density

The first obvious factor to test is the number of tuples, \mathcal{T} , in the base cuboid. Figures 1 - 4 show the four algorithms under low \mathcal{D} and uniform distribution. A couple of observations could be made from them. First, BUC responds linearly as \mathcal{T} increased. This makes sense because most of BUC’s computation times come from counting sort, an $O(n)$ algorithm. Second, Multiway responds sub-linearly to \mathcal{T} . This is mainly because \mathcal{T} does not affect the size of the chunks in Multiway. It only affects the counts in the array indices. Thus the extra times were probably only due to the initial scan of the dataset and final I/O.

The most interesting observation from these 4 figures is that the two H-Cubing algorithms performed progressively worse as \mathcal{C} increased from 5 in Figure 1 to 35 in Figure 4. This reveals an important weakness of the two H-Cubing algorithms. When \mathcal{C} is high and the data is dense, H-Cubing suffers greatly. This is because the H-tree built from the initial data is very wide. Each node has many children and each of those children have many children. In addition, the Header Tables built for the data are big. Searching through them can be cumbersome, even in the presence of a hash table. Precisely for these reasons, the algorithm crawls in Figure 4 compared to BUC and Multiway. Since the data is dense, most aggregate group-bys will meet the minimum support. In fact, in our figures, \mathcal{M} was equal to 1, so all group-bys will satisfy the constraint. Therefore, every single node in the very bushy H-tree needs to be examined.

Before moving onto higher dimensionality data, it would be fair to say that regardless of cardinality, Multiway is the best under low dimensionality, dense data, uniform distribution, and low minimum support.

Now let us examine the algorithms under medium \mathcal{D} in Figures 5 - 9. The same trend from last time continues for the most part. The two H-Cubing methods perform well when \mathcal{C} is low and degrades as \mathcal{C} increases. One interesting observation is that in Figure 9, BUC overtakes Multiway. Several factors are responsible for this. First, \mathcal{M} is 1000 and since BUC performs pruning, it has an advantage. Second, under high \mathcal{D} , the data has now become more sparse vs. Figure 4 because \mathcal{T} has not changed. Under sparse conditions, BUC’s

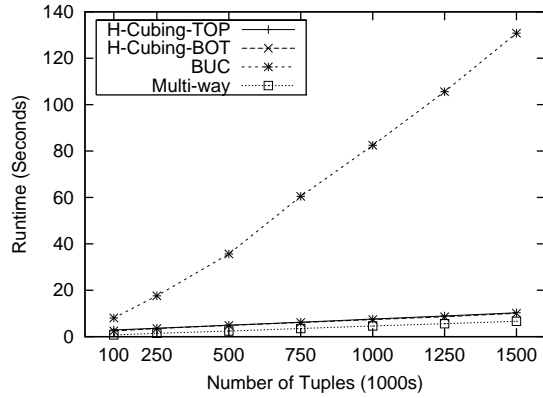


Figure 5: **Tuple Size** $D = 7$, $C = 5$, $S = 0$, $M = 1000$

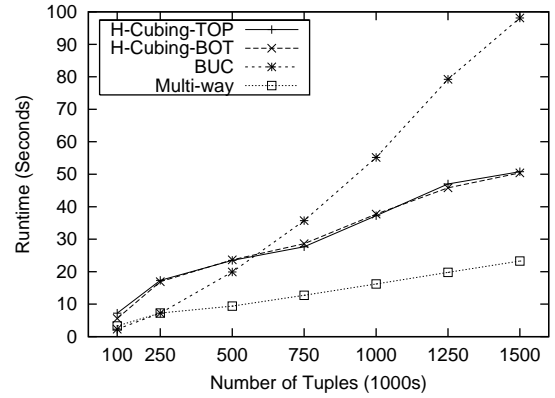


Figure 6: **Tuple Size** $D = 7$, $C = 10$, $S = 0$, $M = 1000$

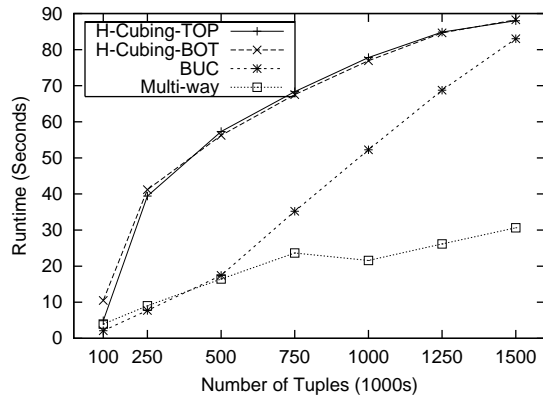


Figure 7: **Tuple Size** $D = 7$, $C = 12$, $S = 0$, $M = 1000$

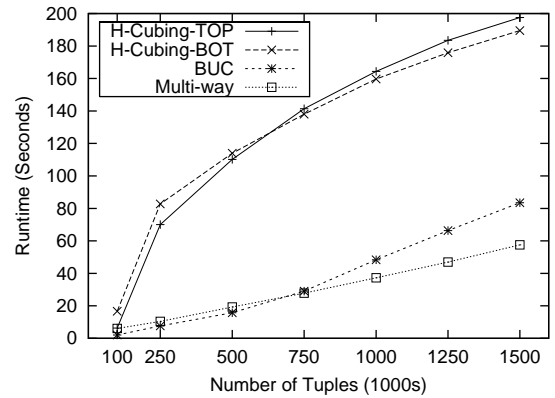


Figure 8: **Tuple Size** $D = 7$, $C = 14$, $S = 0$, $M = 1000$

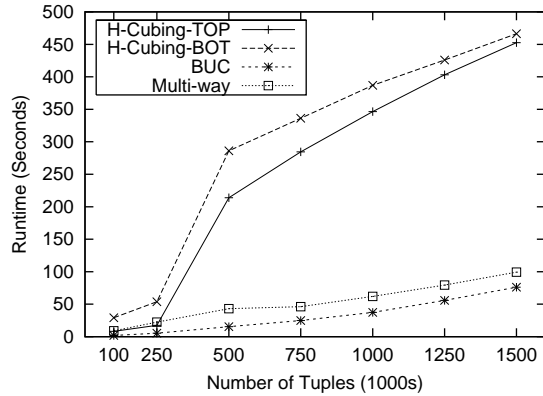


Figure 9: **Tuple Size** $\mathcal{D} = 7$, $\mathcal{C} = 17$, $\mathcal{S} = 0$, $\mathcal{M} = 1000$

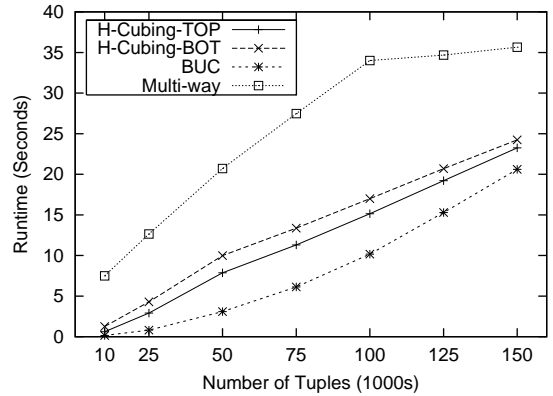


Figure 10: **Tuple Size** $\mathcal{D} = 10$, $\mathcal{C} = 5$, $\mathcal{S} = 0.5$, $\mathcal{M} = 1000$

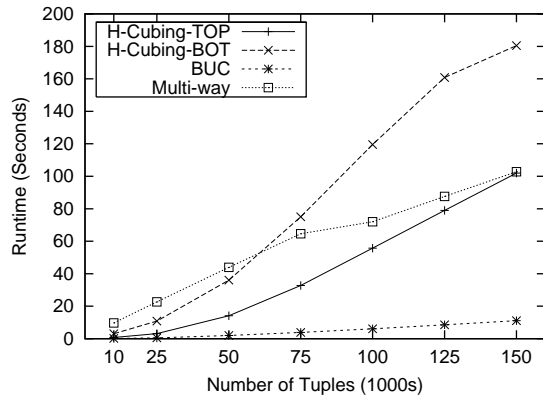


Figure 11: **Tuple Size** $\mathcal{D} = 10$, $\mathcal{C} = 8$, $\mathcal{S} = 0.5$, $\mathcal{M} = 1000$

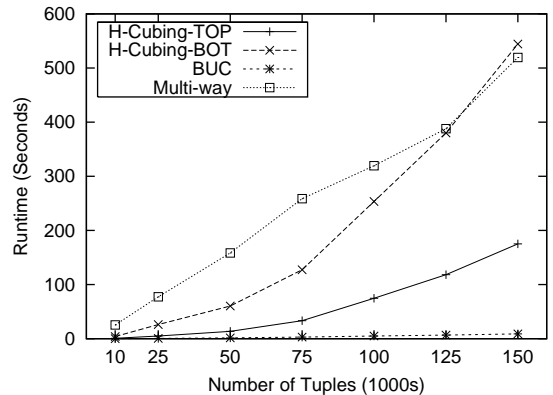


Figure 12: **Tuple Size** $\mathcal{D} = 10$, $\mathcal{C} = 11$, $\mathcal{S} = 0.5$, $\mathcal{M} = 1000$

CountingSort works optimally.

Lastly, we shall examine running times under high \mathcal{D} in Figures 10 to 12. It is now obvious that Multiway performs rather poorly compared to the others. The reason is that Multiway is exponential to \mathcal{D} because all group-bys must be computed. Also, notice that the two H-Cubing algorithms are now performing significantly worse than BUC. This is because the H-tree is much wider and deeper, whereas in BUC, the increase only results in a linear response. It is also notable that H-Cubing BOT becomes worse than H-Cubing TOP when \mathcal{C} gets relatively large.

3.2 Cardinality

A closely related issue to dimensionality and tuple size is cardinality, \mathcal{C} . Let's first examine it under low \mathcal{D} in Figures 13 and 14. All four algorithms increased in running time due to the increase in \mathcal{C} . This is reasonable because there are now more values in each dimension to aggregate. Multiway will have more chunks, BUC will have more to sort, and H-Cubing's H-tree will be wider. The two H-Cubing's response to the increasing \mathcal{C} is the most remarkable;

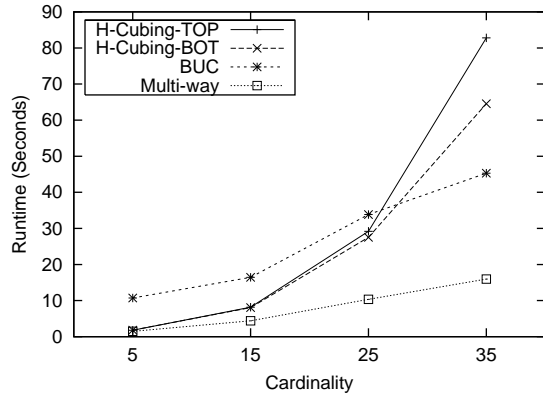


Figure 13: **Cardinality** $\mathcal{D} = 5$, $\mathcal{T} = 500K$, $\mathcal{S} = 0.0$, $\mathcal{M} = 1$

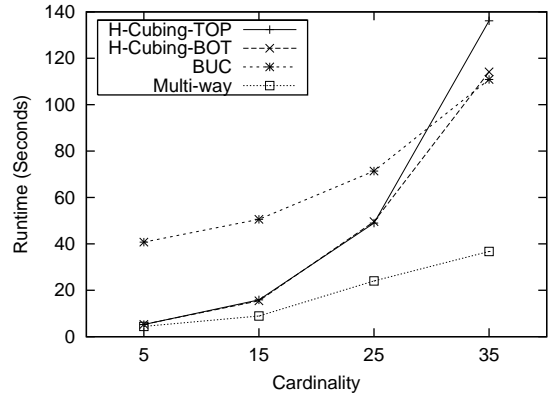


Figure 14: **Cardinality** $\mathcal{D} = 5$, $\mathcal{T} = 1500K$, $\mathcal{S} = 0.0$, $\mathcal{M} = 1$

they give an exponential rise. The reason is that each level in the H-tree is getting wider so the larger \mathcal{C} is affecting the algorithm multiple times.

Let us now examine the same change in \mathcal{C} under medium dimensionality. These can be seen in Figures 15 and 16. First, let us observe Figure 15 where the data is uniform. In this situation, the two H-Cubing algorithms exploded in running time much like last time. The increase in \mathcal{C} compounded with a medium \mathcal{D} is clearly shown. However, it is interesting now that BUC actually improved with the higher \mathcal{C} . This can be attributed to density rather than just \mathcal{C} alone. The data with higher \mathcal{C} is much more sparse, which BUC's CountingSort takes advantage of. The reason this behavior did not occur under low \mathcal{D} was that the data stayed dense even with the higher \mathcal{C} . A similar behavior can be observed again in Figure 16. The reason that the two H-Cubing algorithms did not explode in performance in Figure 16 is due to the high \mathcal{S} . We will discuss this later in the paper.

Lastly, we will check the trend under high dimensionality in Figure 17. As with medium dimensionality, BUC improved due to the sparser data. The two H-Cubing algorithms gave an exponential response because the data was close to uniform. And Multiway gave a linear response, which is reasonable because the modest increase (from 5 to 11) will just increase the number of chunks to process by a small, constant factor.

3.3 Minimum Support Level

With the exception of Multiway, all of the algorithms tested use some form of pruning that exploits the anti-monotonicity of the `count` measure. But how each algorithm does the pruning is different. We shall observe their behavior in Figures 18 - 20.

First, the figures show that Multiway gave no response to a higher \mathcal{M} . Figure 18 shows some speedup, but that's only due to the smaller amount of I/O it needs to do just to output the data cube. In Figures 19 and 20, Multiway remains constant just as we'd expect. BUC and the two H-Cubing algorithms gave speedups when \mathcal{M} is increased. For BUC, this is easily explained because partitions that fail \mathcal{M} will not be explored at all, thus saving computation time. For the two H-Cubing methods, the increased \mathcal{M} has the effect of making

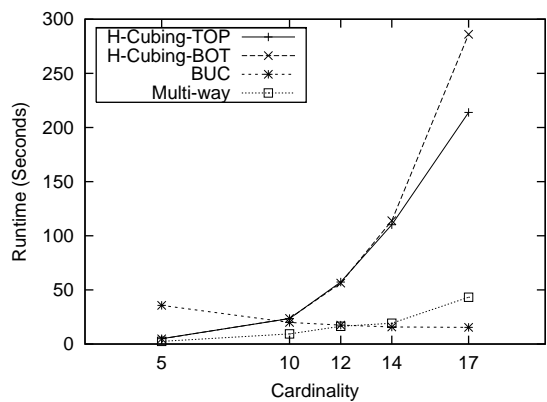


Figure 15: **Cardinality** $D = 7$, $T = 500K$, $S = 0.0$, $M = 1000$

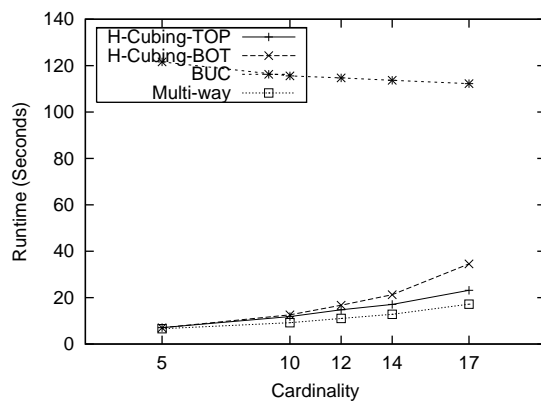


Figure 16: **Cardinality** $D = 7$, $T = 1500K$, $S = 2.0$, $M = 1000$

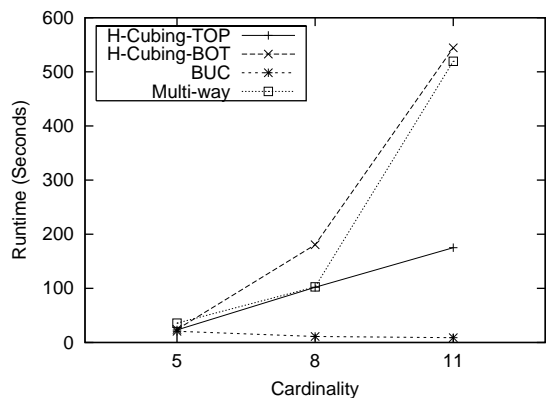


Figure 17: **Cardinality** $D = 10$, $T = 150K$, $S = 0.5$, $M = 1000$

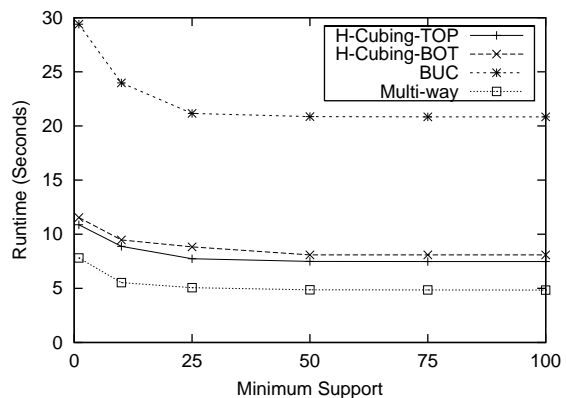


Figure 18: **Minimum Support** $D = 5$, $C = 15$, $T = 1000K$, $S = 0.0$

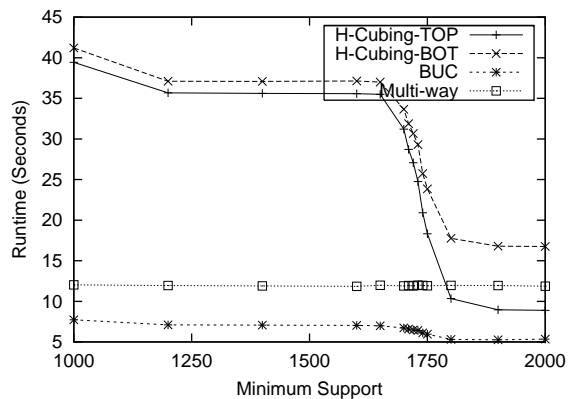


Figure 19: **Minimum Support** $D = 7$, $C = 12$, $T = 250K$, $S = 0.0$

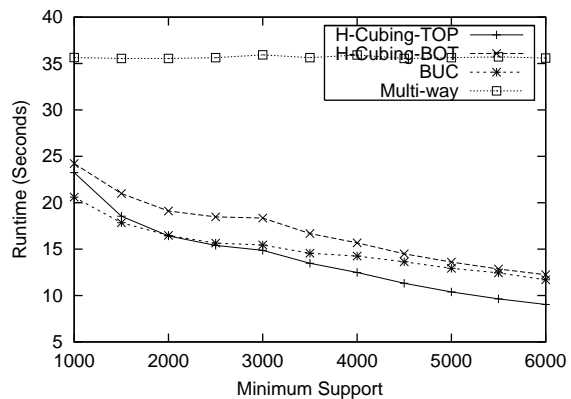


Figure 20: **Minimum Support** $D = 10$, $C = 5$, $T = 150K$, $S = 0.5$

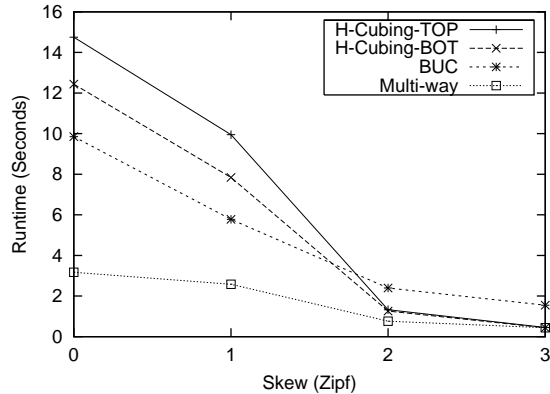


Figure 21: **Skew** $\mathcal{D} = 5$, $\mathcal{T} = 100K$, $\mathcal{C} = 25$, $\mathcal{M} = 1$

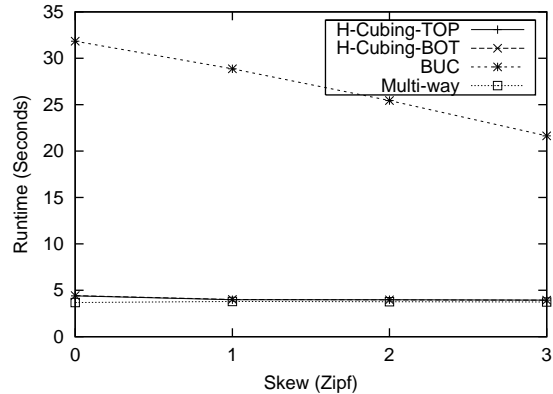


Figure 22: **Skew** $\mathcal{D} = 5$, $\mathcal{T} = 1250K$, $\mathcal{C} = 25$, $\mathcal{M} = 1$

the tree more streamlined. This saves many recursive calls just like BUC. An interesting observation is that in Figure 20, the two H-Cubing algorithms responded better to higher \mathcal{M} than BUC.

3.4 Data Skew

Data skew is a very important factor in all four of the algorithms. See Figure 21 and 22 for how it affects the algorithms under low \mathcal{D} . As the graphs show, under low \mathcal{D} , skewed data makes all four algorithms perform better. One of the reasons is that less distinct aggregate group-bys are generated. This saves the algorithms on I/O. In addition, for low \mathcal{D} and low \mathcal{T} , the data is sparse and fairly compact. For the 2 H-Cubing algorithms in Figure 21, they started performing much better once \mathcal{S} was around 1.5. This can be explained by the size of the H-tree built. With a low \mathcal{D} , the H-tree is already fairly compact because there are only a few levels. With the skewed data, each node in the H-tree is further reduced in size because not all values in each dimension appear now. So as \mathcal{S} increases, the H-tree grew thinner. Thus both of the H-Cubing algorithms were able to overtake BUC. They even approached the levels of Multiway, which we've seen dominates at low \mathcal{D} . In Figure 22, \mathcal{T} is very high, meaning the data is very dense. In this situation, BUC is the only one that improves.

We now turn our attention to data skew under medium \mathcal{D} as shown in Figures 23 and 24. As seen before, H-Cubing becomes much faster in the presence of high skew. The H-tree in this situation is already fairly bushy, and we can see that H-Cubing BOT performed very poorly with $\mathcal{S} = 0.0$. As \mathcal{S} increased, many of the cells in the Header Tables and nodes in the H-tree either disappeared or went under \mathcal{M} . This in effect made the H-tree much more compact and thus we see the significant increase in speed. This phenomenon is even more apparent in Figure 24 for both of the H-Cubing algorithms.

One interesting item is that BUC started to perform worse when \mathcal{S} increased. Before we delve into the specific reasons, let's first examine some more evidence. Figure 25 and 26 shows the changes in the algorithms as \mathcal{S} increases in high \mathcal{D} . In both of these figures, BUC's performance decreased as the data became more skewed. This is most apparent in Figure

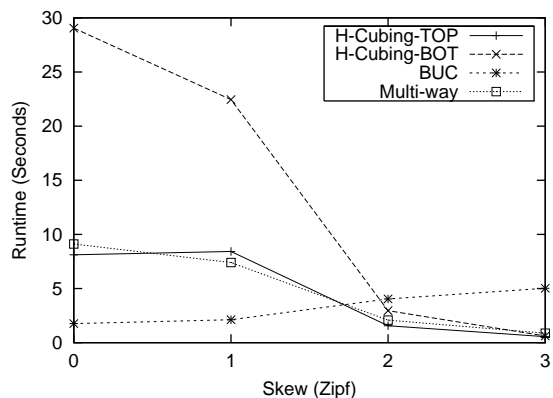


Figure 23: **Skew** $\mathcal{D} = 7$, $\mathcal{T} = 100K$, $\mathcal{C} = 17$, $\mathcal{M} = 1000$

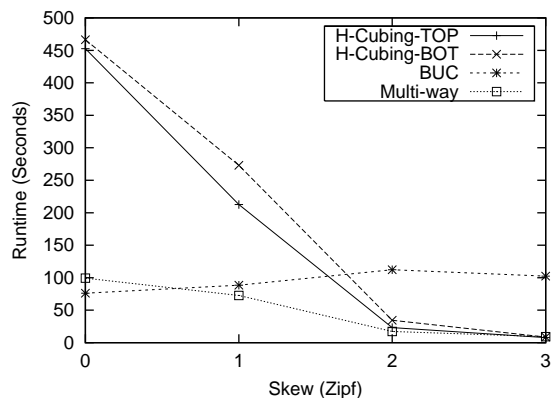


Figure 24: **Skew** $\mathcal{D} = 7$, $\mathcal{T} = 1500K$, $\mathcal{C} = 17$, $\mathcal{M} = 1000$

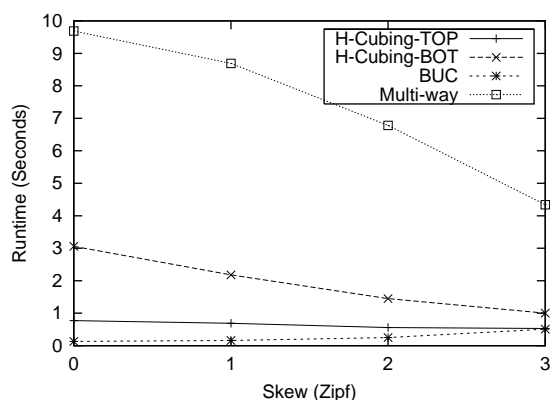


Figure 25: **Skew** $\mathcal{D} = 10$, $\mathcal{T} = 10K$, $\mathcal{C} = 8$, $\mathcal{M} = 1000$

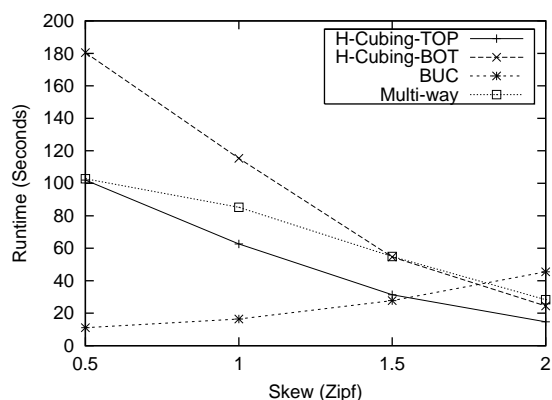


Figure 26: **Skew** $\mathcal{D} = 10$, $\mathcal{T} = 150K$, $\mathcal{C} = 8$, $\mathcal{M} = 1000$

26 where BUC was the fastest at first but moved into last place when \mathcal{S} became 2.0. This is a complete reversal of BUC under low \mathcal{D} . The reason why this is occurring is due to the partitioning and CountingSort. When \mathcal{S} is high, many partitions become fairly small, i.e. the number of tuples are low in that partition. Since BUC recurses on each partition, that small partition will be sorted on the next dimension. The cardinality of that next dimension will be relatively high to the number of values in the partition. Under these conditions, CountingSort performs rather poorly.

In all these tests, Multiway improved as \mathcal{S} increased. Figure 25 shows the most significant change. This is because many chunked arrays now hold a zero counts while other chunks hold a very big count. The array indices with zero count do not need to be process at all while the bigger counts do not increase the workload to Multiway at all. Thus it is able to improve performance.

It is probably fair to say that BUC and H-Cubing TOP are the two best algorithms in high \mathcal{D} . Figure 25 shows that Multiway is slow when the data is more uniformly distributed. And H-Cubing BOT always lagged behind H-Cubing TOP. Let us examine more carefully

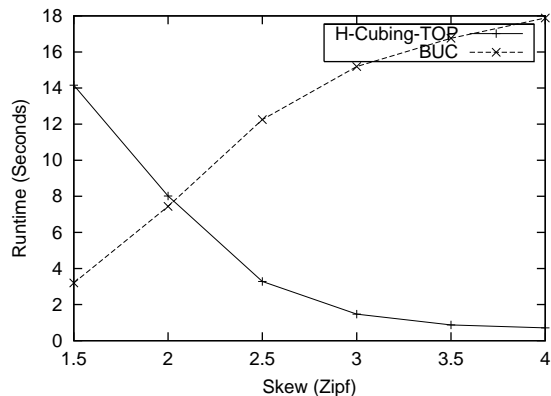


Figure 27: **Skew** $\mathcal{D} = 10$, $\mathcal{T} = 50K$, $\mathcal{C} = 20$, $\mathcal{M} = 1000$

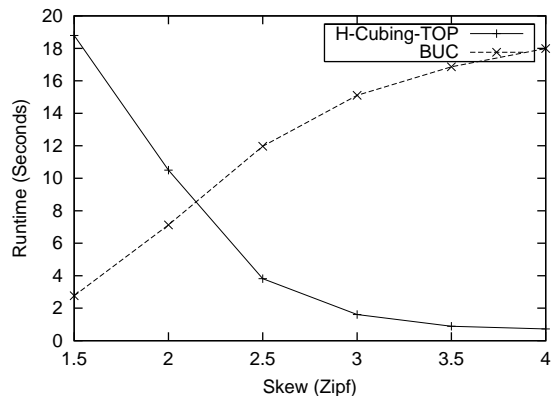


Figure 28: **Skew** $\mathcal{D} = 10$, $\mathcal{T} = 50K$, $\mathcal{C} = 40$, $\mathcal{M} = 1000$

the performances of BUC and H-Cubing TOP in Figures 27 and 28. It is apparent now that when the data is close to uniform distribution, BUC is better. Otherwise, H-Cubing TOP is better. In fact, when the data is purely uniform, $\mathcal{S} = 0.0$, H-Cubing TOP requires an astronomical amount of time.

4 Future Work

Obviously, future work would be to perform more testing. It was difficult to isolate some of the parameters and to test on it alone. More careful methods should be devised to accurately gauge the algorithms. In addition, I/O times should have been removed from the final times to only test the computation times. This is most hurtful during the minimum support level tests. Due to time constraints, some intensive tests involving high dimensionality and high cardinality were avoided (e.g. H-Cubing for 10-dimensional uniform data). It would be helpful to complete all tests. In future tests, changes could also be made to the algorithms. For example, in BUC, it would be wise to implement QuickSort in addition to CountingSort and switch to QuickSort when the data is sufficiently dense. And lastly, it would be very interesting to compare to some other methods, such as *-Cubing.

5 Conclusion

In this paper, we have examined four current methods in computing the Iceberg-CUBE. After 60 some tests, we can only conclude that there is no clear-cut winner. Each algorithm has its advantages and disadvantages under different conditions. For Multiway, it is best when dimensionality is low; however, if the low-dimensional data is skewed, H-Cubing TOP is better. It performs poorly with high dimensionality. For BUC, it is best at high-dimensional data that are sparse and uniform. It performs poorly with high skew and high density. For the two H-Cubing algorithms, they perform well with skewed data and also dense data under low cardinality. They perform poorly to high cardinality. H-Cubing TOP is definitely the better of the two H-Cubing methods. It especially shines when the data is high-dimensional and sparse.

References

- [1] K. Beyer and R. Ramakrishnan. Bottom-up computation of sparse and iceberg cubes. SIGMOD 1999.
- [2] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data Cube: A relational aggregation operator generalizing group-by, cross-tabs, and sub-totals. 1995.
- [3] J. Han, J. Pei, G. Dong, and K. Wang. Efficient computation of iceberg cubes with complex measures. SIGMOD 2001.
- [4] Y. Zhao, P. M. Deshpande, and J. F. Naughton. An array-based algorithm for simultaneous multidimensional aggregates. SIGMOD 1997.

Appendix

For the reader’s viewing pleasure, we have included all other graphs obtained during testing. Some of them are summarized into other graphs earlier in the paper, but they are still very interesting to look at. Enjoy.

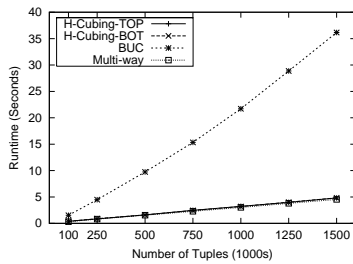


Figure 29: $D = 5, C = 5, S = 1$

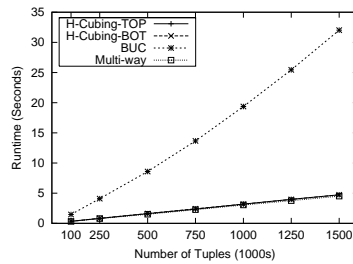


Figure 30: $D = 5, C = 5, S = 2$

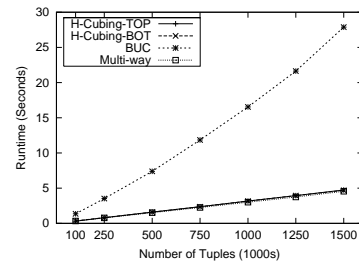


Figure 31: $D = 5, C = 5, S = 3$

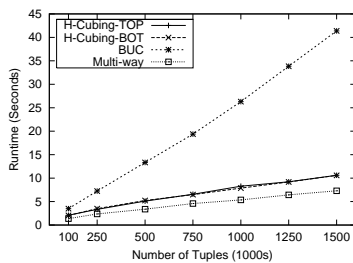


Figure 32: $D = 5, C = 15, S = 1$

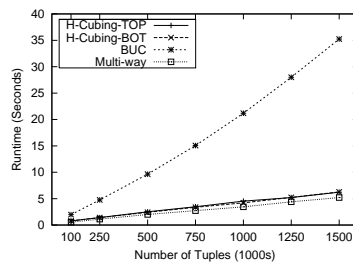


Figure 33: $D = 5, C = 15, S = 2$

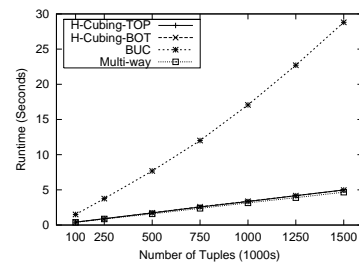


Figure 34: $D = 5, C = 15, S = 3$

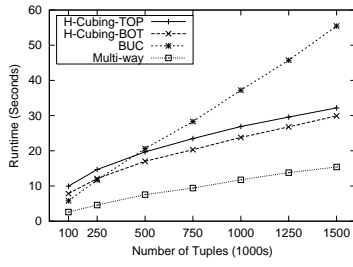


Figure 35: $D = 5, C = 25, S = 1$

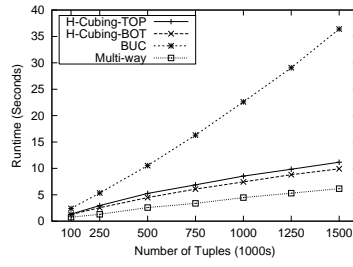


Figure 36: $D = 5, C = 25, S = 2$

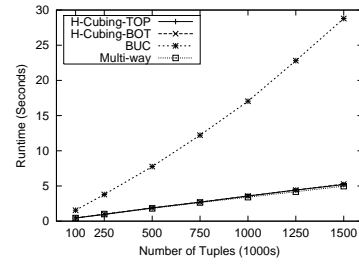


Figure 37: $D = 5, C = 25, S = 3$

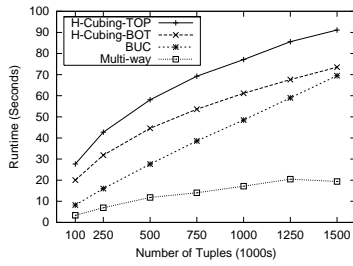


Figure 38: $D = 5, C = 35, S = 1$

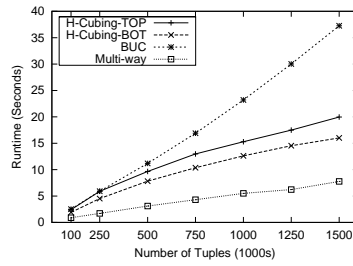


Figure 39: $D = 5, C = 35, S = 2$

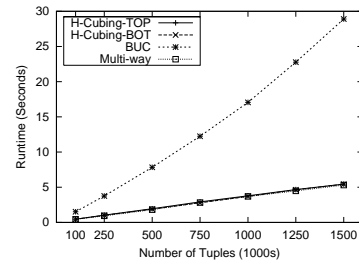


Figure 40: $D = 5, C = 35, S = 3$

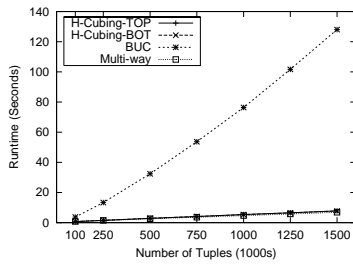


Figure 41: $D = 7, C = 5, S = 1$

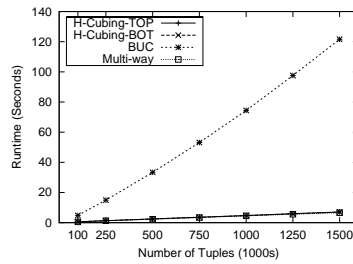


Figure 42: $D = 7, C = 5, S = 2$

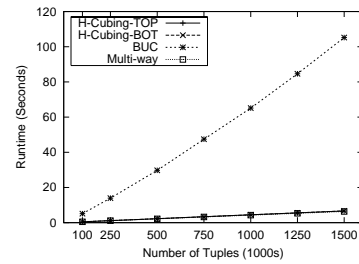


Figure 43: $D = 7, C = 5, S = 3$

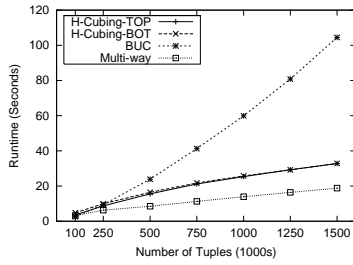


Figure 44: $D = 7, C = 10, S = 1$

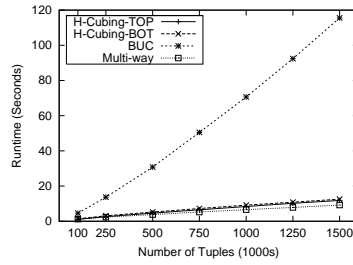


Figure 45: $D = 7, C = 10, S = 2$

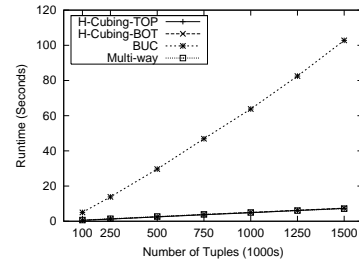


Figure 46: $D = 7, C = 10, S = 3$

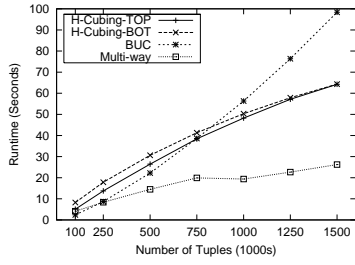


Figure 47: $D = 7, C = 12, S = 1$

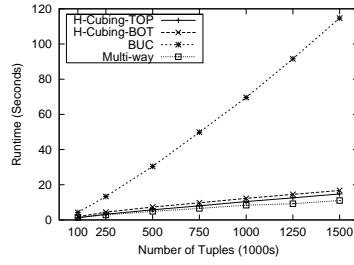


Figure 48: $D = 7, C = 12, S = 2$

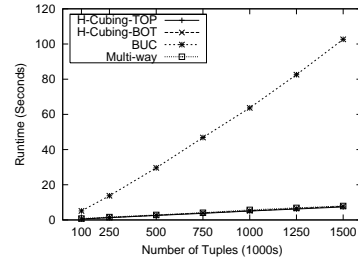


Figure 49: $D = 7, C = 12, S = 3$

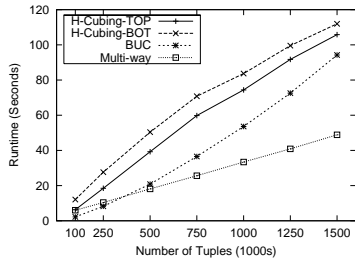


Figure 50: $D = 7, C = 14, S = 1$

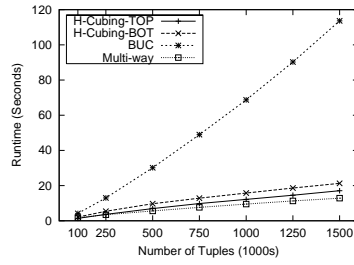


Figure 51: $D = 7, C = 14, S = 2$

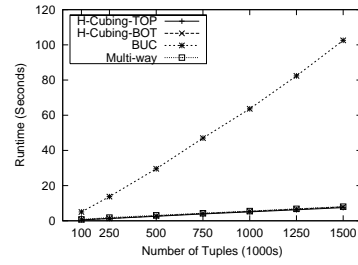


Figure 52: $D = 7, C = 14, S = 3$

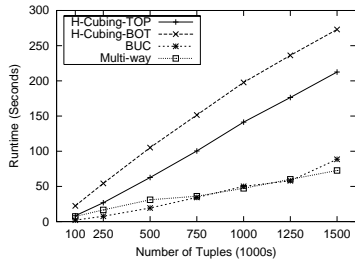


Figure 53: $D = 7, C = 17, S = 1$

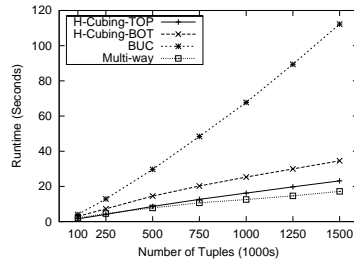


Figure 54: $D = 7, C = 17, S = 2$

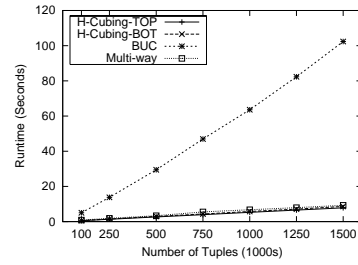


Figure 55: $D = 7, C = 17, S = 3$

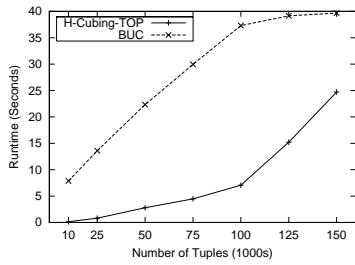


Figure 56: $D = 10, C = 5, S = 0$

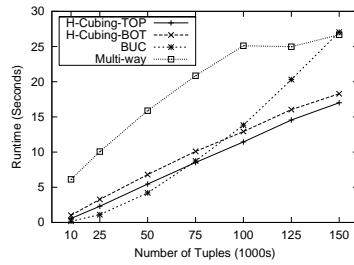


Figure 57: $D = 10, C = 5, S = 1.0$

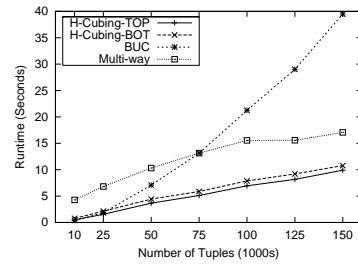


Figure 58: $D = 10, C = 5, S = 1.5$

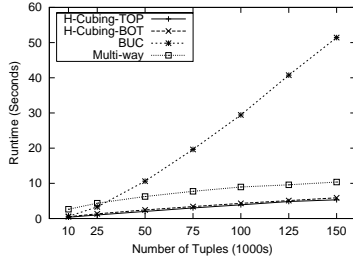


Figure 59: $\mathcal{D} = 10$, $\mathcal{C} = 5$, $\mathcal{S} = 2.0$

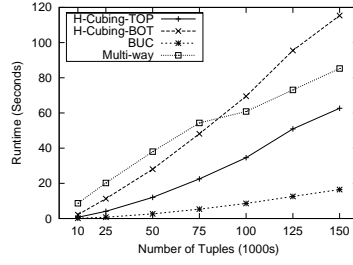


Figure 60: $\mathcal{D} = 10$, $\mathcal{C} = 8$, $\mathcal{S} = 1.0$

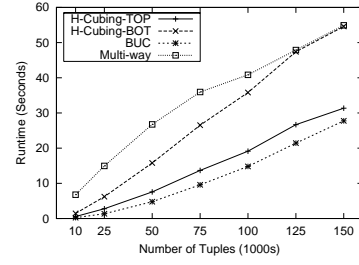


Figure 61: $\mathcal{D} = 10$, $\mathcal{C} = 8$, $\mathcal{S} = 1.5$

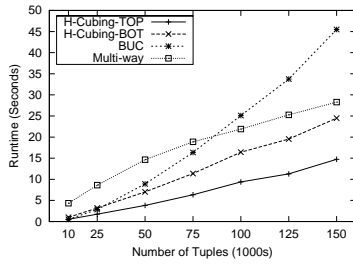


Figure 62: $\mathcal{D} = 10$, $\mathcal{C} = 8$, $\mathcal{S} = 2.0$

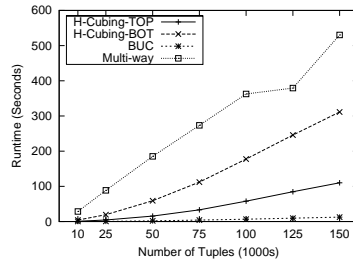


Figure 63: $\mathcal{D} = 10$, $\mathcal{C} = 11$, $\mathcal{S} = 1.0$

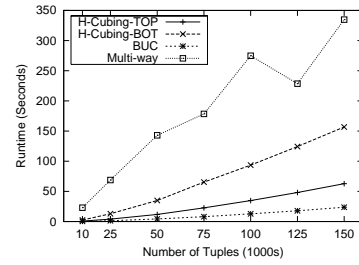


Figure 64: $\mathcal{D} = 10$, $\mathcal{C} = 11$, $\mathcal{S} = 1.5$

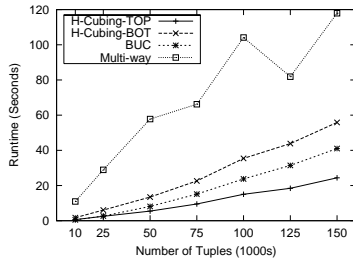


Figure 65: $\mathcal{D} = 10$, $\mathcal{C} = 11$, $\mathcal{S} = 2.0$