Single-round vs Multi-round Distributed Query Processing in Factorized Databases

Lambros Petrou

Wolfson College

University of Oxford

Supervised by Prof. Dan Olteanu

Department of Computer Science, University of Oxford

A dissertation submitted in partial fulfilment

of the requirements for the degree of

Master of Science in Computer Science

Trinity 2015

Acknowledgements

Abstract

Contents

List of Figures

Chapter 1

Introduction

Chapter 2

Preliminaries

Mini TOC

Chapter 3

Finding good Factorization Trees

Mini TOC

Chapter 4

Serialization of Data Factorizations

Mini TOC

Chapter 5

Distributed Query Processing in FDB

Chapter 6

Experimental Evaluation

In this section we will present experimental evaluation for the main contributions of this project, namely the *COST* function for finding good f-trees explained in Chapter 3, the serialization techniques detailed in Chapter 4 and D-FDB, the distributed query engine as presented in Chapter 5.

6.1 Datasets and evaluation setup

This section contains information regarding datasets used and the evaluation setup used to record the reported times and sizes.

6.1.1 Datasets

We used two different datasets throughout the development and evaluation of the above contributions, both described below.

1. *Housing*

   This is a synthetic dataset emulating the textbook example for the house price market.

   It consists of six tables:

   - *House* (postcode, size of living room/kitchen area, price, number of bedrooms, bathrooms, garages and parking lots, etc.)
   - *Shop* (postcode, opening hours, price range, brand, e.g. Costco, Tesco, Sainsbury's)
   - *Institution* (postcode, type of educational institution, e.g., university or school, and number of students)
   - *Restaurant* (postcode, opening hours, and price range)
   - *Demographics* (postcode, average salary, rate of unemployment, criminality, and number of hospitals)
   - *Transport* (postcode, the number of bus lines, train stations, and distance to the city center for the postcode).

The scale factor s determines the number of generated distinct tuples per postcode in each relation: We generate s tuples in *House* and *Shop*, $log2(s)$ tuples in *Institution*, $s/2$ in *Restaurant*, and one in each of *Demographics* and *Transport.* The experiments that use the *Housing* dataset will examine scale factors ranging from 1 to 15.

2. *US retailer*

   The dataset consists of three relations:

   - *Inventory* (storing information about the inventory units for products in a location, at a given date) (84M tuples)
   - *Sales* (1.5M tuples)
   - *Clearance* (370K tuples)

- *ProMarbou* (183K tuples)

6.1.2 Evaluation setup

The reported times for the *COST* function and the serialization techniques were taken on a server with the following specifications:

- Intel Core i7-4770, 3.40 GHz, 8MB cache
- 32GB main memory
- Linux Mint 17 Qiana with Linux kernel 3.13

The experiments to evaluate the distributed query engine D-FDB were run on a cluster of 10 machines with the following specifications:

- Intel Xeon E5-2407 v2, 2.40GHZ, 10M cache
- 32GB main memory, 1600MHz
- Ubuntu 14.04.2 LTS with Linux kernel 3.16

All experiments were run after the application was compiled with optimization flags turned on (i.e. O3, ffastmath, ftree-vectorize, march=native) and with *C++11* enabled.

6.2 COST function – Finding good f-trees

TODO

6.3 Serialization of Data Factorizations

In this section, we evaluate each serialization technique examined and described in Chapter 4. The factorizations we use to evaluate the serialization techniques are the result of applying *NATURAL JOIN* on all the relational tables of the two datasets, *Housing* and *US retailer*.

6.3.1 Correctness of serialization

The correctness test of each serialization was done both in-memory and off-memory (using secondary storage). For equality comparison between two factorizations we use a special function *toSingletons()* that traverses the factorization, encoding the singletons into a string representation that contains *a)* attribute name, *b)* value and *c)* attribute ID in text format, thus creating a huge string that contains the whole data of the factorization.

For the *in-memory* tests we performed the following steps:

1. Load the factorization from disk, let's call it *OriginRep*
2. Serialize it in memory writing into a memory buffer (array of bytes)
3. Deserialize the buffer into a new instance of a factorization, let's call it *SerialRep*
4. Check that the fields of *SerialRep* have valid values
5. Use the *toSingletons()* method and create the string representation for *OriginRep* and *SerialRep* and compare the two strings for equality. This ensures that not only we recover the same number of singletons properly but also that the IDs and values of those singletons are preserved during serialization and de-serialization, even with problematic datatypes like floating point values.

For the *off-memory* tests we performed similar steps as in-memory with an extra additional test to further prove correction.

1. Load the factorization from disk, let's call it *OriginRep*
2. Serialize it to a file on disk (binary file mode)
3. Open the file in read mode and de-serialize it into a new instance of a factorization, let's call it *SerialRep*
4. Check that the fields of *SerialRep* have valid values
5. Use the *toSingletons()* method and create the string representation for *OriginRep* and *SerialRep* and compare the two strings for equality.
6. Enumerate the tuples encoded by the factorizations *OriginRep* and *SerialRep* into two files. Compare the two files for equality using the standard command line tool *diff.*

6.3.2 Serialization sizes

In this section we will examine the size of the serialization output against the flat size of the input factorization (number of tuples).

The **Flat** serialization mentioned in some plots is the simplistic serialization of a flat relational table into bytes. That is by writing the bytes of each value in each tuple one after the other. Therefore, the total size would be equal to $number\ of\ tuples * number\ of\ attributes * 4\ \text{bytes}$ if for example all values are of the data type integer.

Additionally, we used the standard compression algorithms *GZIP* and *BZIP2* to compress *a)* the output serializations and *b)* the flat serialization. We incorporated compression in our experiments to investigate if applying these algorithms on the flat serialization would reduce the size close to our serializations, and also we apply them on the factorization serializations to analyze if there is still improvement to be made regarding value compression as part of our serialization techniques. We will use the notation *GZ1* and *GZ9* to denote compression using *GZIP* at minimum (1) and maximum (9) compression levels respectively. Similarly for *BZIP2* compression using *BZ1* and *BZ9*. The reason we have chosen these two compression techniques is because a) they are widely available and used in almost all web services (e.g. HTTP, REST APIs) and b) *GZIP* is a very fast algorithm with good compression, whereas *BZIP2* is slower but with much better compression, so we can have both choices tested.

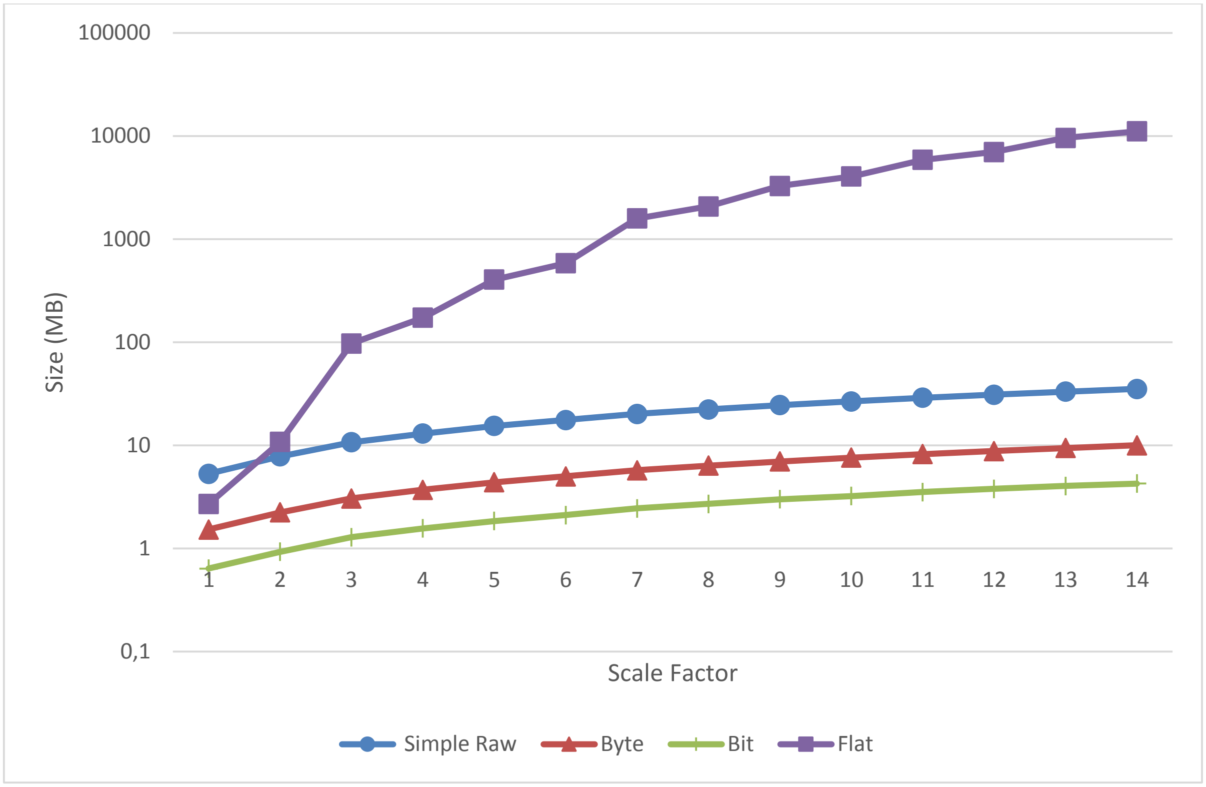

Figure 6.1: Serialization sizes against Flat serialization (Housing)

In Figure 6.1, we present the sizes of the serializations after using each one of our serialization techniques, *Simple* for Simple Raw Serializer, Byte for *Byte* Serializer and *Bit* for Bit Serializer, against the flat serialization for the *Housing* dataset. As expected, the flat serialization size is increasing by several orders of magnitude more than our serializations. This confirms that our serializations retain the theoretical compression factor brought by factorizing the relational table. Moreover, the figure shows that each extension of our serialization brings some additional reduction in the total size with the *Bit Serializer* being the best performing.

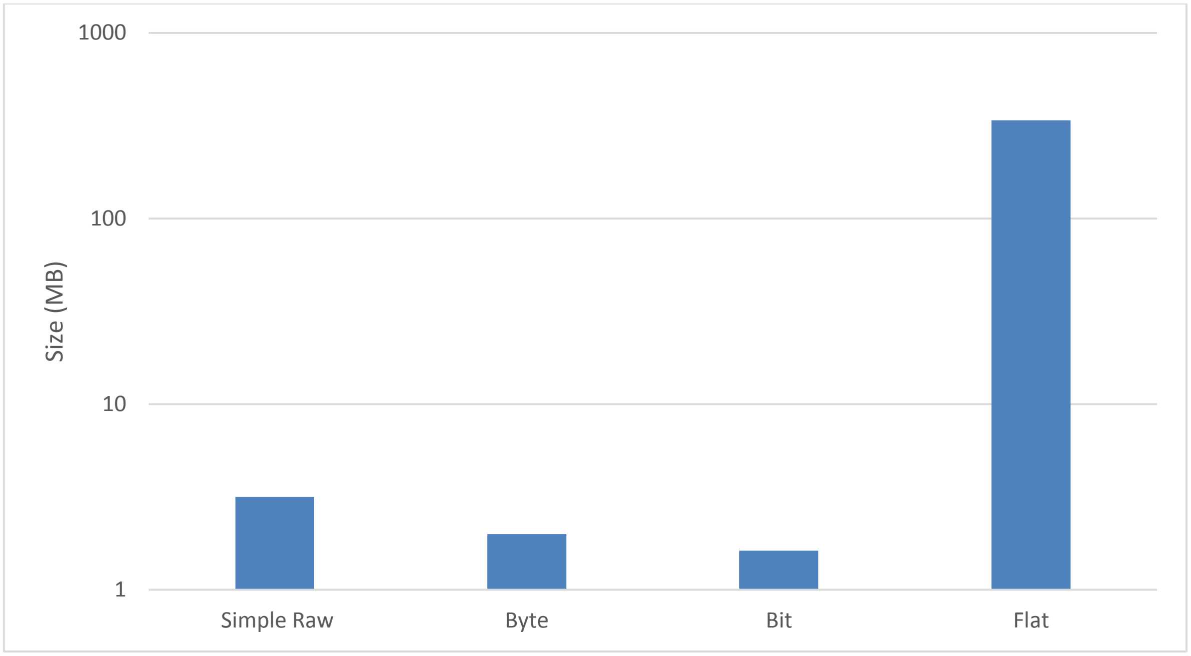

Figure 6.2: Serialization sizes against Flat serialization (US retailer)

The same results are shown in Figure 6.2, where all the serialization sizes follow the same pattern as the *Housing* dataset. The flat serialization is more than two orders of magnitude larger than our fatter serializer, Simple Raw, with Byte and Bit following with smaller output sizes and *Bit* being the best.

In addition, Figure 6.3, presents all three serialization techniques along with compression algorithms applied on their output for additional compression. It is clear that *Simple Raw* serialization which is just the byte enumeration for the values in the factorization grows linearly as the scale factor increases. The second worst serialization is of *Byte Serializer* without any compression applied, but it is very far from the worst and close to the rest of the sizes. A worthy observation is that after applying compression algorithms on-top of *Simple Raw* we get smaller serialization than that of *Byte*'s, which means that the values in this dataset are great candidates for compression. This can be also inferred by the difference in the sizes between the Simple, Byte and Bit outputs since each one uses a more refined technique to use as much less bytes as possible.
Another important point is that Bit serialization is almost perfect, since even when the compression algorithms were applied on it its size did not reduce at all, which means that for this dataset we already do sufficient compression to the values.

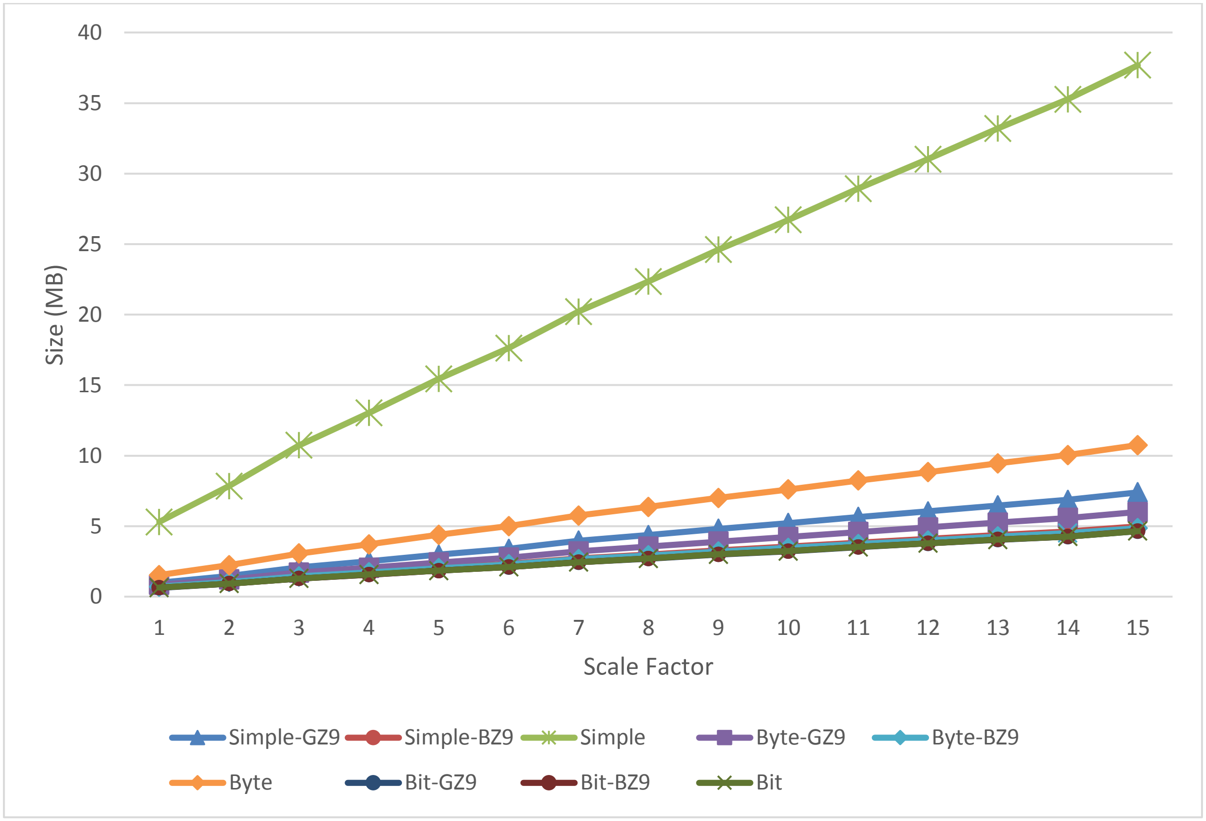

Figure 6.3: Compression GZIP and BZIP2 applied on our serializers

In Figures 6.4 and 6.5 we further explore the effect of additional compression on our serializations. It is clear that the flat serialization can benefit significantly from compression which is expected, but still Figure 6.4 shows that for *Housing* dataset there is a difference between the maximum compression of BZIP2 and GZIP on flat serialization and Bit serialization of two orders of magnitude.

In Figure 6.5 we have different results, which arise due to different datasets. In *US retailer* dataset Bit serialization is still the best performing in terms of output-size but the difference from the flat serialization having applied any of the compression algorithms is not as big as with *Housing* dataset (only around one order smaller). Additionally, the difference between our serializations is also smaller. Having investigated the datasets better, we found that large amount of values in *Housing* dataset are only single-digit numbers, therefore they have a lot leading zero-bits in their representation in memory, hence the big gain using Bit serialization. However, in *US retailer* dataset the values are more random and there are less such values.

Although, the advantage is smaller in *US retailer* using our serialization technique is still more preferable because as we will show later it is considerably faster.

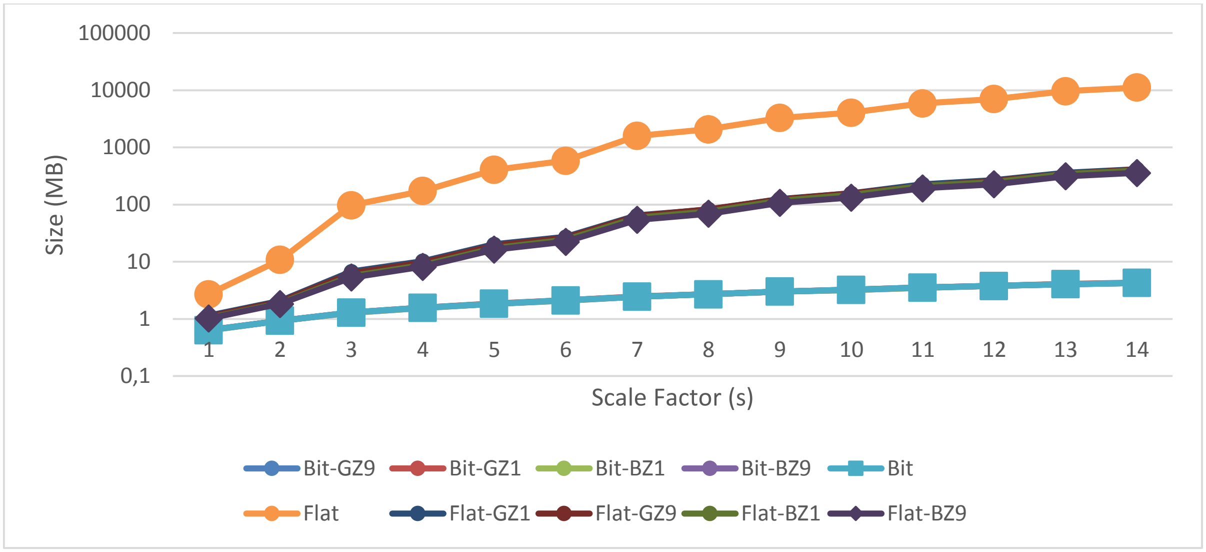

Figure 6.4: Compression GZIP and BZIP2 applied on Bit and Flat serializations (Housing)

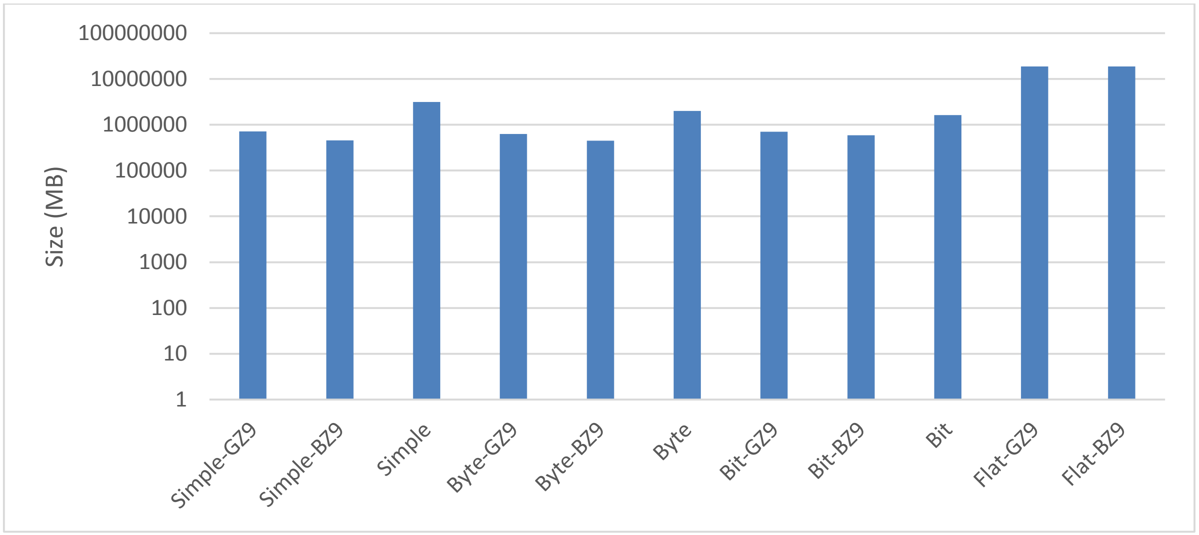

Figure 6.5: Compression GZIP and BZIP2 applied on Bit and Flat serializations (US retailer)

6.3.3 Serialization times

In this section we evaluate the time required to serialize factorizations using our serializers with and without compression techniques on-top.

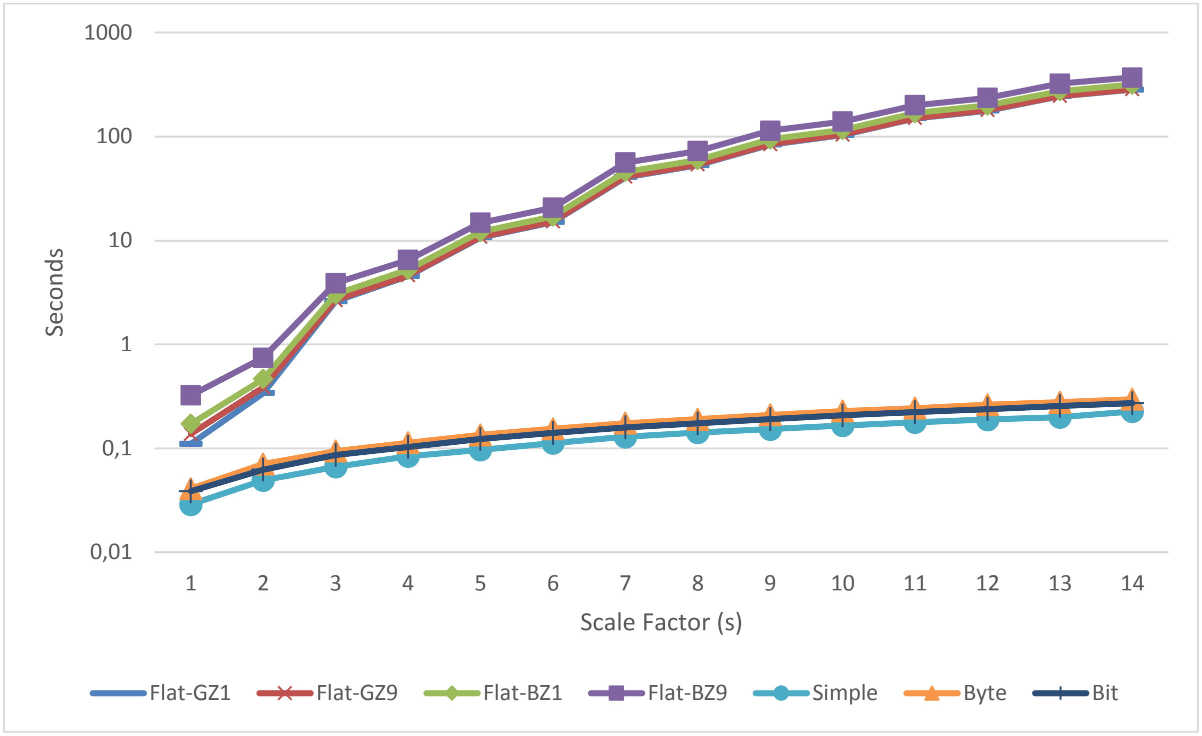

Figure 6.6: Serialization times with compression only on Flat (Housing)

First of all, Figure 6.6, presents the serialization times for our serialization techniques *without* any compression applied and the flat serializations with both compressions. The reason that we decided to show ours without and flat with compression is that we will never ship data over the network *as is* without compressing them due to the huge size, therefore the default choice for a real-world application would be either *GZIP* or *BZIP2* or some other algorithm with similar properties.
The performance of our serialization techniques is more than two orders of magnitude even when applying minimum compression level on flat serialization with both *GZIP* and *BZIP2*.

Figure 6.7 presents the times for all our serializations with and without compression applied on-top. There is significant overhead added, as seen by comparing *Bit* serialization without compression and *Bit-BZ9* for example, or *Byte* with *Byte-GZ9*, but even with compression added the serialization times are significantly faster than compressing the flat serialization.

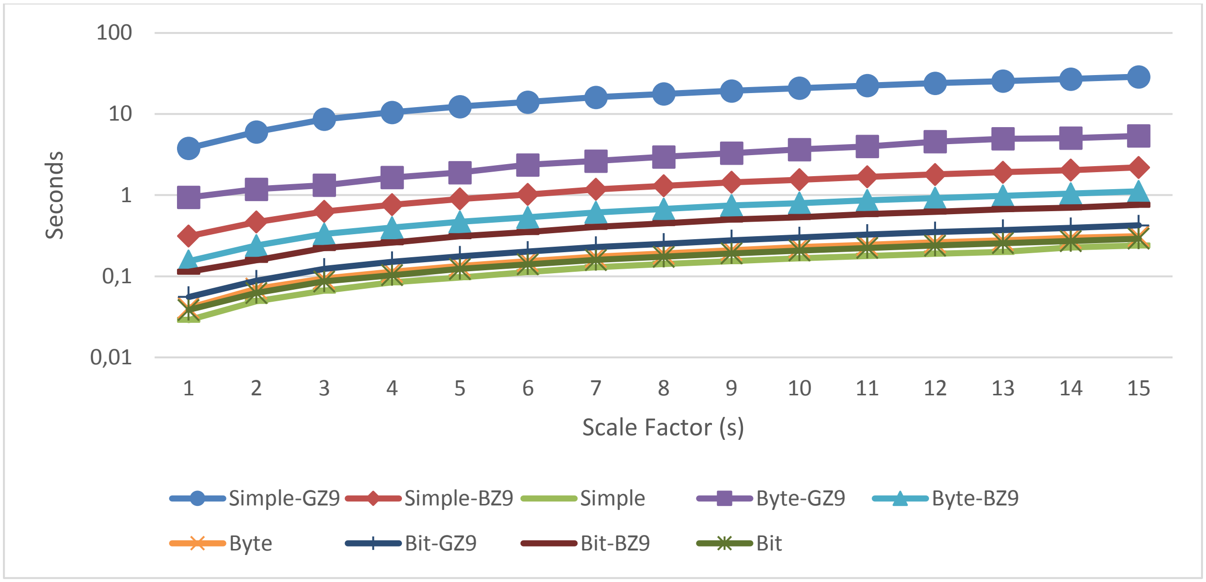

Figure 6.7: Serialization times with compression (Housing)

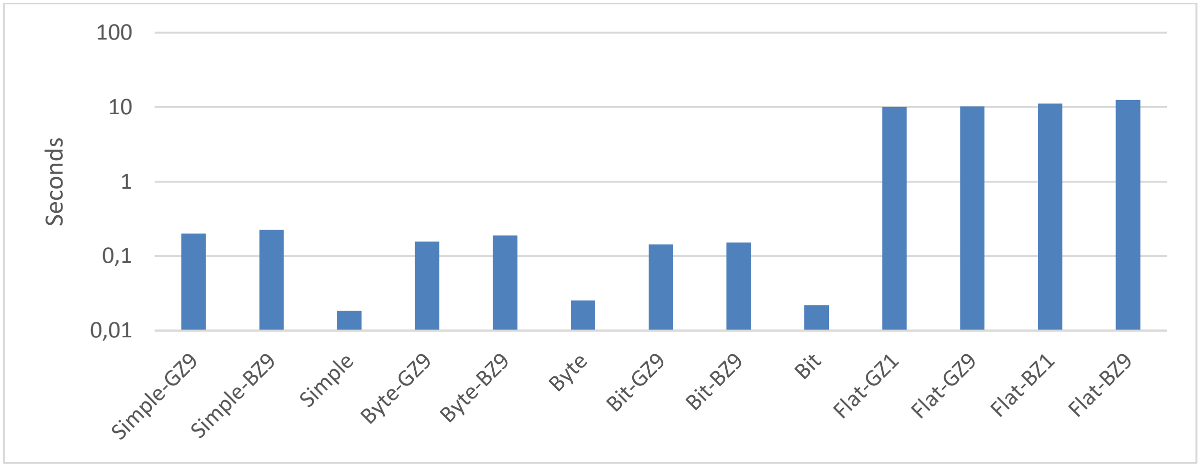

Figure 6.8: Serialization times with compression (US retailer)

Figure 6.8, shows the same experiment, compression applied on-top of the serialization and we see very similar results. Compression upon the flat serialization is a lot slower than compression upon our serializations, which in turn is slower than our serialization without compression.

6.3.4 Deserialization times

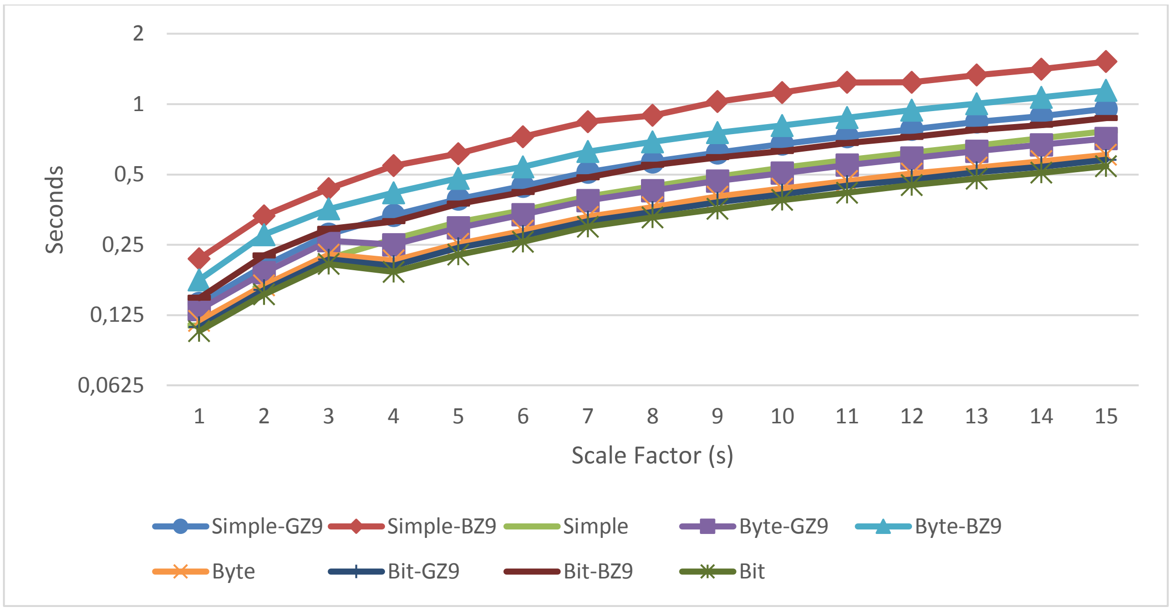

Figure 6.9: Deserialization times for our de-serializers (Housing)

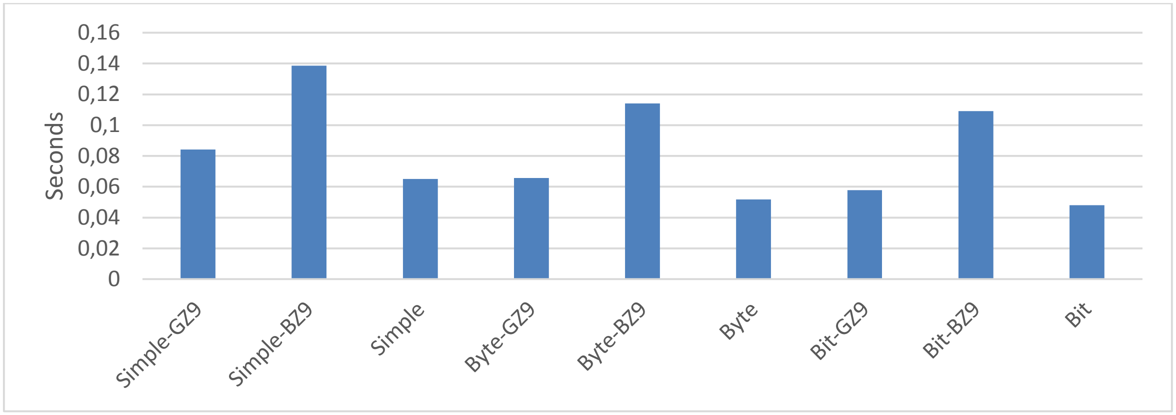

Figure 6.10: Deserialization times for our de-serializers (US retailer)

In this section we examine the time needed to de-serialize a serialized factorization back into a factorization in memory.

Both datasets have similar results, as seen in Figures 6.9 and 6.10. It is obvious that *BZIP2* is the slowest in all three de-serializers. *GZIP* compression is fast and this is shown in our results since the difference between de-serializing with and without this compression is small, however it is still an overhead. It is remarkable to that even though *Byte* and *Bit* have additional complexity compared to *Simple Raw* de-serializer they both have faster times, which is due to the smaller total size they process.

6.3.5 Conclusions

We performed a variety of experiments with all three serializations against two datasets with different characteristics (one artificial with a lot of single-digit values, one real-world dataset with complex values). We also compared our serialization against the flat serialization with and without compression.

Analyzing the results of these experiments led to the following conclusions:

- The three serializers retain the theoretical compression of factorizations against flat relational tables into their serializations.
- The flat serialization requires significantly more time to apply compression on its data than our serializers with and without compression applied on them.
- The benefit of applying additional compression over the three serialization techniques depends mostly on the actual factorization values, but especially with *Bit Serializer,* which is the final version, it is questionable whether the additional overhead to compress is worthy.
- We showed that it would be very interesting to explore additional extensions to Bit Serializer in order to enhance its compression capabilities. A very important feature of our serialization algorithms is that during de-serialization we *do not have to* process *all* the data as is the case with standard compression algorithms that process large blocks each time.
- Overall, we conclude that Bit Serializer can be the basis of more advanced serialization techniques for factorizations and that even at this stage it can be a great alternative to standard compression algorithms for systems that use factorizations as a means of data communication.

Chapter 7

Conclusions and Future Work

A player faces a dynamic optimization problem of 5 periods. Let a_t denotes the player's action in period t,

$$a_t \in \{P, N\} \tag{1}$$

We denote the vector of action choices by $\boldsymbol{a} = (a_1, a_2, a_3)$. Playing in a period yields an immediately consumption level of x at a certain future cost, to be paid at period 4, while not playing yields no consumption and incurs no cost, so

$$x_t = \begin{cases} x & if\ a_t = P \\ 0 & if\ a_t = N \end{cases} \tag{2}$$

The player observe x in period 1 before she pick her action.
Let C_s denotes total cost for playing s games and S_t the number of games played up till and including time t.

This paper.[1] Theoretically, ...
The issue of ...

[1] Ashraf et. al [1] uses a ...

This paper is organized as follows. The next section presents ... Then, Section 3 discusses
the ... Section 4 analyzes the ... Concluding remarks are offered in Section 5.

References

Ashraf, Nava, Dean Karlan and Wesley Yin. “Tying Odysseus to the Mast: Evidence from a Commitment Savings Product in the Philippines.” Quarterly Journal of Economics. Vol. 121, No. 2, pp. 635-672. May 2006.