**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

Chapter 1 Introduction 1

Chapter 2 Preliminaries 2

Chapter 3 Finding good Factorization Trees 3

Chapter 4 Serialization of Data Factorizations 4

Chapter 5 Distributed Query Processing in FDB 5

Chapter 6 Experimental Evaluation 6

6.1 Datasets and evaluation setup 6

6.1.1 Datasets 7

6.1.2 Evaluation setup 8

6.2 COST – Finding good f-trees 8

6.3 Serialization of Data Factorizations 11

6.3.1 Correctness of serialization 12

6.3.2 Serialization sizes 13

6.3.3 Serialization times 17

6.3.4 Deserialization times 20

6.3.5 Conclusions 21

Chapter 7 Conclusions and Future Work 22

Appendix A 24

References 30

List of Figures

**No table of figures entries found.**

# Introduction

|  |
| --- |
| Mini TOC |

# Preliminaries

|  |
| --- |
| Mini TOC |

# Finding good Factorization Trees

|  |
| --- |
| Mini TOC |

## Motivation

Previous work on Factorized Databases (\*\*REFERENCE HERE\*\*) provides searching for a good factorization tree (f-tree) upon a database based on asymptotic bounds and the size of the input. It has been proven to be optimal, many times generating exponentially more compressed representations than normal flat relational databases.

Although complexity bounds are nice, there are a lot of cases where they are not sufficient and we need more explicit properties. For example, given a database Q, the previous work might find that the optimal f-tree has parameter *s(Q)* = 2, where *s(Q)* is the cost measurement function, and that there are multiple trees with this property. But the question is which of those f-trees having parameter *s(Q)* = 2 is better ? At the moment, the implementation just uses the *first* f-tree that has the optimal parameter.

What we really want to investigate is how to find a good f-tree, using more refined parameters, that will also depend on the *data* we want to factorize and not only on the f-tree structure which ignores data (except relation sizes). The reason why this is an important part of the project is that in a distributed system, \*\*see discussion in experiments Section X.Y\*\*, the biggest bottleneck is communication and data distribution. Therefore, although *s(Q)* provides optimal trees we want to minimize communication cost, thus requiring an f-tree that results in the smallest factorization size possible.

For example, in real-world scenarios it can happen that two f-trees have the same *s(Q)* parameter, let's say 2, but they might differ in size with a factor of 4x. More precisely, f-tree A can produce a factorization with 1 million singletons (value nodes) where f-tree B can produce a factorization of 4 million singletons. Asymptotically, we cannot discriminate the two, but in real life using f-tree B will result in excessive data distribution thus increasing our communication cost a lot, so it does matter in the end-to-end processing.

## Contribution

This chapter's contribution is a *COST* function that given an f-tree and certain statistics (number of unique values per attribute, number of unique values per attribute under any other attribute of the f-tree) returns an estimation of the total factorization size (number of singletons, value nodes) that would occur if our database (factorization) was factorized based on that given f-tree.

## Idea

The requirement is to have a cost function that would take into account the actual values of a database instance in order to be able to compare in a more precise manner f-trees that are asymptotically optimal.

Let's start with some facts about FDB factorizations:

1. each union has its values ordered in ascending order
2. each union has unique values
3. a factorization may have many relation dependencies and each dependency forces its attributes to exist along a single path in the f-tree (like a linear linked list)
4. some attributes belong to many relations, thus have many dependencies

Considering the above facts, we used the number of unique values per union, therefore easily finding unique values per attribute. Additionally, the dependencies matter a lot since in complex queries like *triangles* or *squares*, see Figure 3.2, we have all the attributes in a single path, forming a single linked list and each level down the path affects the factorization size.

Figure 3.1: Example f-tree

Figure 3.2: Triangle and Square queries



Figure 3.3: Example factorization

We define *cost* of a factorization the total number of value nodes or singletons, thus the sum of the number of value nodes for each attribute. For example the factorization in Figure 3.3 has 20 value nodes (black nodes) so the cost for that f-tree is 20.

### Initial thoughts

A first idea was to use an f-tree as a reference tree and based on some statistics calculated on this reference tree we would calculate the factorization estimated size for any other arbitrary f-tree.

Given an f-tree and its factorization, we calculate for each attribute the average number of unique values (children of a union) under any of its ancestor attributes. The average is taken over all the ancestor's children values.

**Notation**

1. *XuY* denotes the average number of unique values of attribute X *under* a single value of attribute Y, where Y is an ancestor of X.
2. *uniq(X)* denotes the average unique number of values among all the unions of attribute X.

For example, assuming the f-tree in Figure 3.1 and its factorization, see Figure 3.3, we have the following statistics:

* *unique values per attribute:* uniq(A), uniq(B), uniq(C), uniq(D), uniq(E), uniq(F)
* *number of unique values per attribute X under an ancestor attribute Y:*   
  BuA, CuA, CuB, DuA, DuB, EuA, FuA, FuE

Having the above statistics calculated, given any other f-tree *T* the estimated size of the factorization would be calculated by summing the estimated number of nodes for each attribute. To calculate the cost for an attribute X, a path between X and its parent in T should be found inside the reference tree, followed by the multiplication of all the pair-wise averages (*XuY*) along the path to get an estimation for the number of values of X.

This approach quickly turned out to be wrong and over-estimating because of the excessive usage of estimates when we multiplied them for all the attribute pairs along the path.

### Proposed Idea

The final solution is based on the same intuition but in a more precise and more accurate way. Instead of depending on estimates of a reference tree which lead to artificial over-estimation, statistics such that we can use them with any f-tree should be calculated, regardless of the input f-tree. Recall that our *cost* function should be able to accept an arbitrary f-tree and return the estimation size as accurate as possible.

As a result, the following properties (statistics) are used during estimation:

1. *Average number of unique values of attribute X under any attribute Y* (single value of Y), denoted as *XuY*, where Y is an ancestor of X.
2. *Average unique number of values among all union nodes for each attribute,* denoted as *uniq(X)* where X is an attribute.
3. Flat size of the database (number of tuples).

Another important observation is that the number of nodes for each attribute in the factorization is related to *all* of its ancestor attributes and not only to its parent. For example, in figure 3.1, the number of nodes for attribute C depend both on B *and* A, therefore we somehow have to incorporate them in our estimation for attribute C.

In the following formula *COST(X)* denotes the estimated number of value nodes (singletons) for attribute X in the result factorization.

Input:

1. f-tree T
2. *XuY* and *uniq(X)*, as described above
3. flat factorization size

|  |  |
| --- | --- |
| **Estimation Formula**  If attribute X is f-tree root:  Else:  Where: *MIN\_AVERAGE(X, T)* = the minimum average *XuY*, where Y is an ancestor of X along the path from X to the root of f-tree T. Y should also exist in a common relation with X (dependency). |  |

The above formula gives an estimation for the number of value nodes for a given attribute in a given factorization tree. The total size of the factorization is the sum of the individual cost for each attribute.

It is important that we take into consideration dependencies and only use *XuY* averages for the ancestor attributes that are in a common relation with attribute X since we do not know the relationship of X with attributes in other relations.

Additionally, we restrict the estimation size of the number of values per attribute to the flat size of the representation since that is the maximum amount of singletons we can have for each attribute, which is the worst case where each tuple is a separate path in the factorization.

## Algorithms

In this section the pseudocode for the complete factorization size estimation procedure is provided that implements the *COST* function described above.

**Estimate Factorization Size**

The algorithm is an iteration over the attributes in the factorization tree in a BFS-traversal order memoizing the estimations of already visited attributes to use in their descendants cost calculation.

|  |  |
| --- | --- |
| **Algorithm 3.1:** Calculate estimated size for factorization using given f-tree | |
|  | // @fTree: the f-tree to estimate the size for, if used for factorization  // @FLATSIZE: the flat size in number of tuples  double estimate\_size(FactorizationTree \*fTree, unsigned int FLATSIZE) {  // queue for BFS - holds pairs of attribute IDs <parentID, childID>  queue<pair<int, int>> Q;  // memoization array of costs estimated - size = number of attributes  vector<double> costs(ftree->num\_of\_attributes());  // cost for the root  rootID = fTree->root->ID;  costs[rootID] = uniq(rootID);  // add root's children in queue  **for each** child attribute CA in fTree->root->children {  Q.push\_back({rootID, CA->ID});  }  **while** (!Q.empty()) {  parent\_child = Q.pop\_front();  parentID = parent\_child->first;  childID = parent\_child->second;    // calculate the minimum of all averages XuY where X = childID and  // Y is every ancestor of X in the fTree that belongs to a common  // relation (dependency) with X.  double min\_est = min\_average(fTree, childID);  // calculate the cost for this attribute  // COST(X) = min(COST(par(X)) \* min(all averages XuY), FLAT\_SIZE)  costs[childID] = min((costs[parentID] \* min\_est), FLATSIZE);  // add the attribute's children to the BFS queue  **for each** child attribute CA in fTree->node(childID)->children {  Q.push\_back({childID, CA->ID});  }  }  // the total cost estimation is the total number of value nodes  // which is the sum of all the value nodes for each attribute  return sum(costs);  } |

Algorithm 3.1 calculates the estimated size of the representation that will be created based on the input factorization tree. The algorithm assumes that the averages are already calculated and are ready to be used. This is common in the databases-world where some properties are calculated off-line in order to be used during runtime (value histograms, unique values, selectivity, etc.).

The complexity of the algorithm is quadratic to the number of attributes in the factorization tree, since we visit each attribute exactly once and for each attribute we call the *min\_average()* function which has linear complexity, or more precisely its complexity depends on the longest root-to-leaf path (we visit each attribute's ancestor in the f-tree).

For the sake of completion the code for *min\_averages()* function is provided below.

|  |  |
| --- | --- |
| **Algorithm 3.2:** Find min *XuY* for an attribute | |
|  | // @fTree: the factorization tree we currently estimate the size  // @attributeID: the attributeID we are calculating the estimated number of nodes  **double min\_average(FactorizationTree \*fTree, attributeID)** {  // get the attribute node  cN = fTree->node(attributeID);  // the maximum average for each attribute is the unique number of values of it  double min\_est = uniq(attributeID);  // we now traverse the path from the current attribute up to the root  // and check the average of children with each ancestor  // ONLY if it belongs to common relation/dependency (hyperedge)  **while** (cN != NULL) {  if (same\_hyperedge(attributeID, cN->ID)) {  min\_est = min(min\_est, XuY(attributeID, cN->ID));  }  cN = cN->parent;  }  return min\_est;  } |

The complexity of the above pseudocode is linear in the longest path from an attribute node to the root and it finds the minimum average number of children (unique values) of the current attribute under any ancestor attribute in the current f-tree.

The maximum amount of children (unique values) of any attribute under any other attribute is the amount its unique values since we have unique values in our union nodes.

**Calculate averages**

The previous algorithm that estimates factorization size assumes existence of the averages *XuY* for each pair of attributes in the same *hyper-edge* (relation/dependency). A procedure was implemented that calculates this but it is code-specific to be included in the thesis so we only provide a pseudocode for it showing the idea behind it.

The function returns a two-dimensional matrix with size (, where is the number of attributes. *Matrix[X][Y]* corresponds to the notation used above, *XuY*, which means that cell located at row X and column Y has the average number of children (unique values) among all unions of attribute X which are located below each value of the attribute Y.

|  |  |
| --- | --- |
| **Algorithm 3.3:** Calculate averages | |
|  | // @fTree: the factorization tree used for the representation '@fRep'  // @fRep: an input factorization of the database instance we examine  **double[][] calculate\_averages(FactorizationTree \*fTree, FRepresentation \*fRep)** {  double matrix[fTree->number\_of\_attributes()][fTree->number\_of\_attributes()];  **for each** attribute A in fTree->nodes {  // make the current attribute A root of the factorization  make\_root\_attribute(A, fRep, fTree);  // traverse the factorization in either DFS or BFS mode and calculate  // all the averages where attribute A is the parent since now all  // other attributes are below attribute A  averages = calculate\_averages\_for\_root(A, fRep);  update\_matrix(matrix, averages);  }  return matrix;  } |

The above algorithm's runtime could be improved but it is orthogonal to the project and only used during the off-line pre-processing of the database instance to generate the averages, thus its sub-optimality is not a serious concern.

The real need was to provide a fast cost function that during runtime could determine the size of the factorization given an arbitrary f-tree.

# Serialization of Data Factorizations

|  |
| --- |
| Mini TOC |

# Distributed Query Processing in FDB

|  |
| --- |
| Mini TOC |

# Experimental Evaluation

|  |
| --- |
| 6.1 Datasets and evaluation setup 6  6.1.1 Datasets 7  6.1.2 Evaluation setup 8  6.2 COST – Finding good f-trees 8  6.3 Serialization of Data Factorizations 11  6.3.1 Correctness of serialization 12  6.3.2 Serialization sizes 13  6.3.3 Serialization times 17  6.3.4 Deserialization times 20  6.3.5 Conclusions 21 |

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.

## Datasets and evaluation setup

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

### 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 tuples in *House* and *Shop*, tuples in *Institution*, 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.

1. *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)

### 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.

## COST – Finding good f-trees

In this section we evaluate the *COST* function, analyzed in Chapter 3. Through the following experiments we try to decide whether having a function that *estimates* the factorization size, in number of singletons, using statistics (i.e. unique values per attribute, number of unique values of attribute under another attribute’s single value) derived from off-line preprocessing will give us better insights on f-tree selection compared to the existing work that uses the theoretical size bounds of FDB, parameter *s(Q)*.

For this experiment we will only use the *Housing* dataset for which we devised the optimal f-tree by hand, let’s call it *Tree-O*. Recall that *Housing* dataset JOINs six relations on their common attribute *postcode*.

The optimal f-tree *Tree-O* is as shown in Figure 6.1.



Figure 6.1: Optimal f-tree for Housing

This f-tree has a parameter and it is optimal, which means FDB cannot find any f-tree asymptotically better than *Tree-O*. In order to evaluate our *COST* function we change the order of some attributes in their relation paths and compare the real factorization size in number of singletons with the estimation by *COST*.

The biggest desire here is for the estimations to follow the trend of real size with each f-tree, if not predict exactly. All these f-trees have therefore they are indistinguishable by FDB.

From the experiments we made we noticed that while the scale factor increases the behavior of the *COST* differs. We will present results for Housing scale factors 1, 5 and 9 that show this variance in accuracy and ten different f-trees. The f-trees can be found in Appendix A.

In Figures 6.2, 6.3 and 6.4 we present the relation between real size and estimated size for each of the 11 f-trees we examine (optimal and its varieties) for scale factors one, five and nine respectively. In small datasets, see Figure 6.2, we see that the *COST* function estimates exactly the number of singletons in the factorization, which is the same for all f-trees. This leads us to believe that the branches in the factorization become single paths very early and the COST restricts its estimation by the total size of a relation, hence always matching the real case.

Figure 6.2: Real vs COST (Housing - 1)

Moving to scale factor five, see Figure 6.3, we see that the real size now differs up to 150 000 singletons among some f-trees. The estimated size is very high sometimes due to excessive usage of averages which can be misleading in many cases. However, we can see that the trend of the estimated size follows the real size which shows that it could be very useful to at least be able to eliminate very bad f-trees, always among those that have the optimal *s(Q)* parameter.

Figure 6.3: Real vs COST (Housing - 5)

Really interesting is the fact that f-tree 9 is always estimated wrongly by *COST*, which shows the weakness in using global averages, specifically the number of unique values of an attribute X under any other attribute Y. F-Tree 9 modifies the optimal f-tree in its fourth subtree (sainsburys*, …, openinghoursshop*) such that it is completely reverse, thus having *openinghoursshop* on top, as child of *postcode*.

Figure 6.4: Real vs COST (Housing - 9)

Similar results can be seen in Figure 6.4, where again the *COST* function overestimates the size with some f-trees. F-Tree 1 swaps *house* with *flat* and f-tree 2 takes *house* as leaf of its branch as seen in the optimal f-tree.

In conclusion, although the COST is overestimating with some f-trees it can be used to reject some *bad* f-trees.

## 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*.

### 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*.

### 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 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 6.5: Serialization sizes against Flat serialization (Housing)

In Figure 6.5, 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 6.6: Serialization sizes against Flat serialization (US retailer)

The same results are shown in Figure 6.6, 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 6.7, 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 6.7: Compression GZIP and BZIP2 applied on our serializers

In Figures 6.8 and 6.9 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 6.8 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 6.9 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 many leading zero-bits in their representation in memory (sometimes 31 out of 32), 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 6.8: Compression GZIP and BZIP2 applied on Bit and Flat serializations (Housing)

Figure 6.9: Compression GZIP and BZIP2 applied on Bit and Flat serializations (US retailer)

### Serialization times

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

Figure 6.10: Serialization times with compression only on Flat (Housing)

First of all, Figure 6.10, 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 6.11 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 6.11: Serialization times with compression (Housing)

Figure 6.12: Serialization times with compression (US retailer)

Figure 6.12, 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.

### Deserialization times

Figure 6.13: Deserialization times for our de-serializers (Housing)

Figure 6.14: 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 6.13 and 6.14. 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.

### 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.

# Conclusions and Future Work

|  |
| --- |
| Mini TOC |

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

|  |  |
| --- | --- |
|  |  |

We denote the vector of action choices by . 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

|  |  |
| --- | --- |
|  |  |

The player observe *x* in period 1 before she pick her action.

Let denotes total cost for playing *s* games and the number of games played up till and including time *t*.

This paper.[[1]](#footnote-1) Theoretically, ...

The issue of ...

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.

|  |  |
| --- | --- |
| **Algorithm 3.1:** Calculate estimated size for factorization using given f-tree | |
|  | // @fTree: the f-tree to estimate the size for, if used for factorization  // @FLATSIZE: the flat size in number of tuples |

# Appendix A



Appendix 1: F-Tree 1



Appendix 2: F-Tree 2



# 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.

1. Ashraf et. al [1] uses a ... [↑](#footnote-ref-1)