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

First of all, I would like to express my greatest appreciation to my project supervisor, Prof. Dan Olteanu, for his guidance, advices and feedback he provided me with the last couple of months. I really enjoyed our discussions on the different aspects of the project, many times occurred during midnight hours chatting online or during all-day meetings! He supported me the whole time and was always available when I needed him.

Furthermore, I would like to thank my family and friends inside and outside of the university circles that were near me and helped me successfully finish this degree each one in his/her own way. I want to explicitly express my gratitude for my mother, since she was always there for me, supporting me, financially and emotionally, to keep moving forward and aiming for the best throughout my whole life.

In addition, a special thank you to *University of Oxford* that funded me this whole year by nominating me for one of the HEFCE scholarships, which without it I would not have been able to be here studying at Oxford.

Finally, I would like to thank the famous *Victoria's Secret* for its amazing shows, "*Victoria's Secret Fashion Shows*", for the countless hours of multimedia content which allowed me to think out-of-the box when things got ugly or when I needed a brain-reboot, during the long programming sessions demanded by this project.

Abstract

Blah Blah...

Contents

List of Figures

Chapter 1

Introduction

Mini TOC

Chapter 2

Preliminaries

Mini TOC

Chapter 3

Finding good Factorization Trees

3.1 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.

3.2 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.

3.3 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

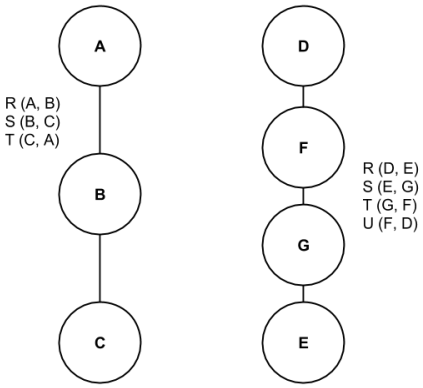


Figure 3.2: Triangle and Square queries

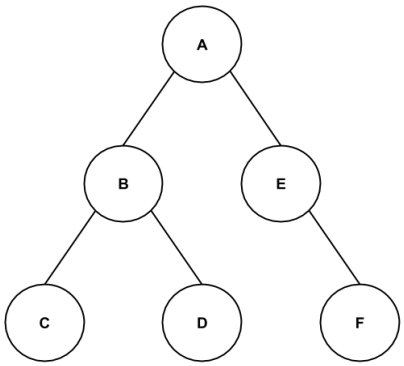


Figure 3.1: Example f-tree

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.

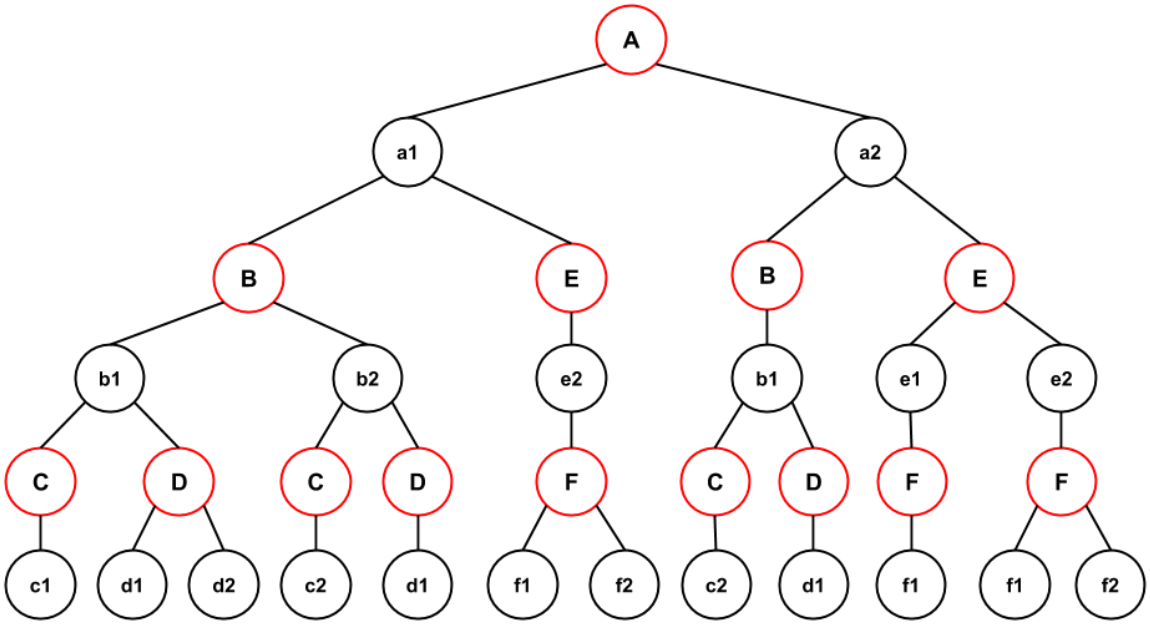


Figure 3.3: Example factorization

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.

3.3.1 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) $X u Y$ 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.

3.3.2 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:

$$COST(X) = uniq(X)$$

Else:

$$COST(X) = MIN(\, COST(parent(X)) * MIN\_AVERAGE(X,T), FLAT\_SIZE \,)$$

$$\text{COST(X)} = \begin{cases} uniq(X) & if \text{ X tree root} \\ \min(\text{COST}(\text{parent(X)}) * \text{MIN\_AVERAGE(X, T)} \,, \text{FLAT\_SIZE}) & if \text{ X internal} \end{cases} \quad (1)$$

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.

3.4 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 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, $O(N^2)$ 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).

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);
```

```
}
```

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 (N x N, 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.

Chapter 4

Serialization of Data Factorizations

4.1 Motivation

An important part of the project investigated ways to serialize, and possibly compress, factorizations (f-representations). It is very important to support serialization and deserialization of a factorization both in a centralized setting and in a distributed setting. For example, sometimes we want to save an instance of a database on disk to manipulate and further process it later. In some other cases we want to ship data over the wire to neighboring nodes which need the data for additional processing on their side.

In general, serialization is the method of efficiently converting an in-memory factorization into a byte stream which is stored or transferred and later can be deserialized into the exact source factorization.
An important aspect of serialization and deserialization is that they have to be efficient in both processing time and space since we want to retain the major benefit of factorizations, which is the compression factor compared to corresponding flat representations. Thus, having a serialization that would take a lot of space or requiring a lot of time to process would be inappropriate for our setting, especially for the distributed system that is the goal.

4.2 Contributions

The contributions made to the project out of this chapter are four (4) serialization techniques for Data Factorizations and one (1) serialization technique for Factorization Trees, namely:

1. *Factorization Tree (De)serializer* - this is the only serialization technique for f-trees and is used in conjuction any of the factorization serialization techniques.
2. *Simple Raw (De)Serializer* - a simple serialization technique that is fast and retains the compression factor over flat representations.
3. *Byte (De)Serializer* - an extension to the Simple serialization technique to only store the required number of bytes for each value.
4. *Bit (De)Serializer* - a further extension to Byte serialization to only store the required number of bits for each value, with specialized methods that can be extended in the future to better support more values to allow better compression.
5. *Bit Serializer HyperCube* - the serialization technique that is used in the distributed system which differs from the normal *Bit Serializer* in that it does not ship all the values of a union but only those that should be shipped based on the Dimensions ID given (this will be explained thoroughly in Chapter 5).

During the preliminary stages, I also implemented a Boost-based serialization technique using *Boost::Serialization* library but it has been abandoned because it turned out to be very bloated and did not satisfy our requirements (explained in-detail later).

4.3 Factorization Serializations

In this section I will describe the different approaches I have taken for the serialization leading to the final version used in the distributed system.

4.3.1 Example

Let us first define an example scenario that we use throughout the chapter.

Assume that we started with four (4) relations, *R(A,B,C)*, *S(A,B,D)*, *T(A,E)* and *U(E,F)*, and applied a *NATURAL JOIN* operator on all of them, resulting in the final table shown below.

A	B	C	D	E	F
1	1	1	1	2	1
1	1	1	1	2	2
1	1	1	2	2	1
1	1	1	2	2	2
1	2	2	1	2	1
1	2	2	1	2	2
2	1	2	1	1	1
2	1	2	1	2	1
2	1	2	1	2	2

Figure 4.1: Result after Natural JOIN on R, S, T, U relations

Show the relation tables too // TODO

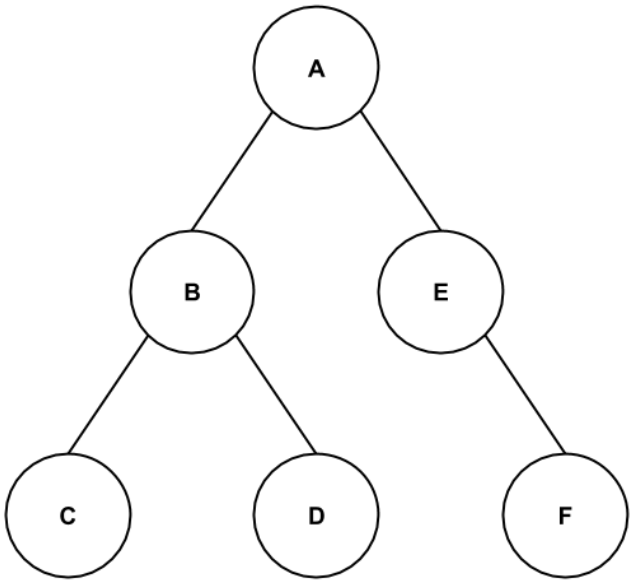


Figure 4.2: Example f-tree

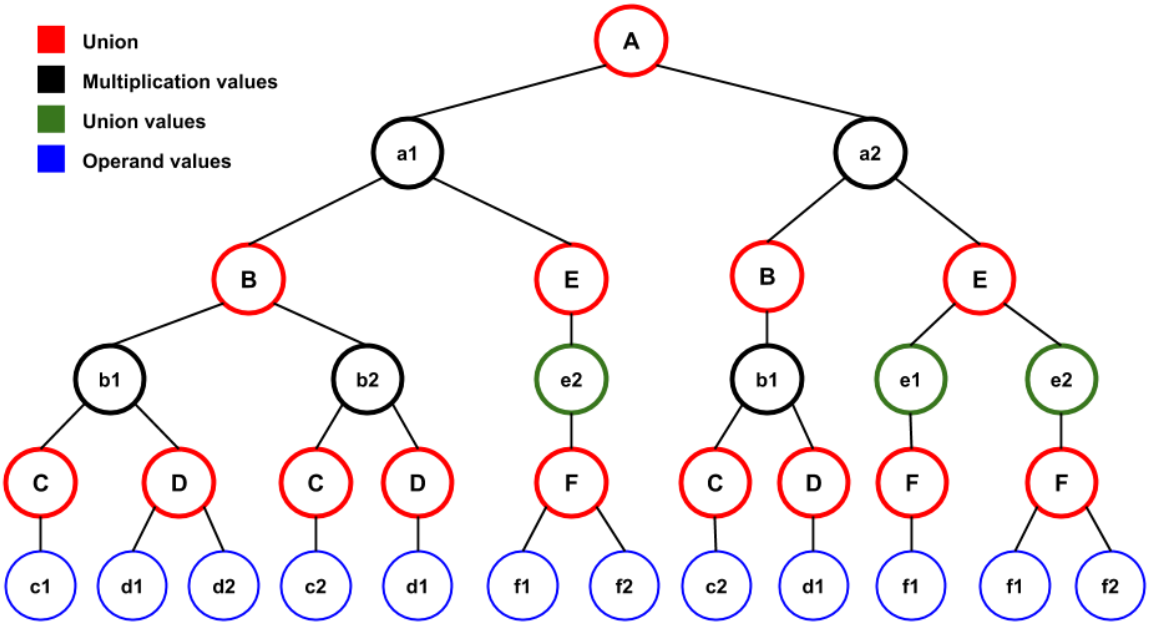


Figure 4.3: Example factorization

We will use the f-tree in Figure 4.2, named *Example f-tree* to factorize the result of the JOIN (see relational table in Figure 4.1) with the Data Factorization based on that f-tree being shown in Figure 4.3.

I will abstractly explain the in-memory representation of Data Factorizations as implemented at the moment. A factorized representation contains the following types of nodes:

1. *Union* nodes just contain a list of the values for that specific attribute union
2. *Multiplication values* are value nodes that act like *Multiplication* nodes since their attribute is a multiplication attribute (based on f-tree) and they have two or more Union nodes as children
3. *Union values* are value nodes that just have one Union node as a child
4. *Operand values* is just another node type to denote leaf values

4.3.2 F-Ttree serialization

An f-tree is the back-bone component of a factorization since it defines the structure of the representation and all the relations between the attributes of the query.
The serialization of an f-tree is the same for all the different factorization serialization techniques and is implemented as a separate module since it is a very small data structure (a few KBs) and we do not mind using the simplest serialization for it.

We use the same structure as an f-tree definition file for its serialization. As a result, the serialization of the f-tree show in Figure 4.2 is as follows:

```
6 4
A int
B int
C int
D int
E int
F int
-1 0 1 1 0 4
R S T U
```

```
2 3 4 5
A,B,C
A,B,D
A,E
E,F
```

The first line defines the number of attributes N and relations M in the f-tree, followed by N lines declaring the name of each attribute and its data type. The current FDB implementation assigns IDs to the attributes in the order they are defined here with the first attribute (A in this example) being given ID zero (0) and the last attribute (F in this example) being given ID five (*N-1*).
The next line defines the tree-relationship since for each attribute we specify the ID of its parent attribute. A has parent ID -1 which means A is root, then B has parent ID 0, C has parent ID 1, D has parent ID 1, E has parent ID 0 and F has parent ID 4.

Then we similarly have a line containing the relation names, again being given IDs internally, with the following line specifying for each relation its parent attribute node. The last M lines are just the relations enumeration with their attributes.

The serialization of an f-tree uses **Text** format and it is prefixed with its size length to allow the deserializer to know up-front the total f-tree serialization size in order to read all the information at once (prefixing messages with their total size is very common in message passing protocols).

The serialized f-tree (including its size header) is prefixed in the final serialization of the Data Factorization such that it can be deserialized first and allow us to use it during the factorization deserialization.

4.3.3 Boost Serialization

As a first attempt to provide serialization/deserialization we decided to use *Boost::Serialization* library since it gathered high rating reviews among the online community and since *Boost* was already being used for the networking modules of the system it seemed to be a great fit.

The purpose of *Boost::Serialization* library is to allow developers to provide an easy way to add serialization to their *existing* data structures without writing a lot of boilerplate code since it can be described more or less like a memory dump of a data structure into a stream (file, socket, etc.).

The integration of the library in FDB and the actual implementation was pretty straightforward. There were some *special* methods required to be added in each class we wanted to be serializable according to certain library rules. However, the end result was really disappointing due to very big serialization size.

As I mentioned, this is more of a memory dump of the structure, including any pointers and their destination objects, in order to easily allow the deserializer to create the exact data structure. The major problem here and the reason of the bloated serialized output is that the existing *FDB* implementation is not as space-efficient as it should be and that overhead is transferred into the serialization.

The current data structure of a factorization has a lot of overhead, like keeping all the values of a Union as a Double-Linked-List thus introducing excessive amount of pointers. As a result, the serialization module was dumping everything, more importantly the pointer references, to allow re-creation during deserialization leading to a bloated outcome, both in terms of raw size in bytes but also in long serialization times.

In our first preliminary experiments the serialized representation was almost the same size as the flat-relational representation, thus completely eliminating the compression factor of FDB over flat databases, which was unacceptable!

In order to use *Boost::Serialization* and at the same time having quality serialization we had to write custom code for each implementation class for every data structure we use to omit certain fields or doing my own book-keeping for the pointers and references to avoid all this going into the serialized output. Eventually, this was not worth it since Boost was still going to add some overhead which cannot be removed, like class versioning.

The *first attempt failed* but led to some interesting observations. Although the current implementation was poorly done, a good serialization does not need all that information and we could also take advantage of the special structure of a factorization to make it as succinct as possible.

4.3.4 Simple Raw (De)Serializer

Before going into details for this serialization technique we state some observations made after investigating the reasons that led to failure of the previous attempt for serialization.

- Each factorization is strictly associated with a factorization tree (f-tree) that defines its structure.
- The main types of a node in a factorization are the *Multiplication* (cross product) and the *Summation* (union) node types.
- The values inside a union node can be stored in continuous memory, thus avoiding the excessive overhead of Double-Linked-Lists due to the pointers for each value.
- There is necessity to de-couple the data, values, from the factorization structure since a lot of overhead comes with the representation and not the data.

Apart from the above observations, the trick that led to this serialization method is that the only nodes required to be serialized are the *Union* nodes along with their values. Since each factorization strictly follows an f-tree, it became obvious to use the f-tree as a guide during serialization and deserialization leading to a more succinct outcome which just contains the absolute minimum of information, *the values!*

The problem with generic serialization techniques, like *Boost* described above is that all information goes into the serialized outcome to allow correct deserialization. We can avoid this overhead in our case since we know the special structure of the factorization and therefore we can use the f-tree to infer the structure of the representation and load the values from the serialized form as we go along during deserialization.

4.3.4.1 Idea

The main idea of *Simple Serializer* is that we traverse the factorization in a DFS (Depth-First-Search) order and every time we find a *Union* node we serialize it, then recurse to the next. The serialization of a union node is simple and just contains a number N indicating the number of values in that specific union, followed by N values of the attribute represented by that union.

For example, if a specific union of attribute A (of type *int*) has the values *[3, 6, 7, 8, 123, 349]*, its serialization would be:

6 3 6 7 8 123 349

It is important to mention that we use **Binary** read and write methods during serialization and deserialization and for each children count we use 32-bit unsigned integer values whereas for the actual values the corresponding number of bytes required for that attribute data type (i.e. *double* = *sizeof(double)* = 8 bytes) is used.

The serialization of a factorization is just a sequence of *children counts* followed by their *corresponding values*. The important benefit of this serialization technique is that only the absolute minimum information required to recover the representation is stored.

Moreover, *Simple Serializer* assumes that we already deserialized the f-tree (discussed previously) and we can use it to infer the structure of the representation.

4.3.4.2 Algorithms

Algorithm 4.1: Simple Serializer

```
// @node: the starting node of our serialization (initially root of factorization)
// @fTree: the factorization tree to be used as guide
// @out: the outpout stream to write the serialization
dfs_save(Operation *op, FactorizationTree *fTree, ostream *out) {
    if (is_multiplication(op)) {
        // in multiplication nodes we just recurse without serializing
        for each child attribute CA in op->children { dfs_save(CA, fTree, out); }
    } else if (is_union(op)) {
```

```
        // in union we serialize the number of children and values
        write_binary(out, op->childrenCount);
        // serialize union values
        for each child value V in op->children { write_binary(out, V); }

        // recurse only if the union's attribute is not leaf in the f-tree
        if (!is_leaf_attribute(fTree, node->attributeID)) {
            for each child value CV in op->children { dfs_save(CV, fTree, out); }
        }
    }
}
```

Simple Serializer, see Algorithm 4.1, is an extension of the well-known DFS traversal algorithm for trees with in-order value processing.

The representation has two types of nodes, thus leading to two different treatments in serialization. When a *multiplication* node is encountered, the algorithm recurses on its descendants without serializing anything since the multiplication information can be inferred from the f-tree. When a *union* node is encountered we first serialize the number of values in that union, followed by the serialization of all the values. At the end we recurse on each of child to complete the DFS-traversal.

The algorithm iterates over the values twice since all the values of a union have to be serialized completely and *then* move on to the next union, like in an in-order traversal. Additionally, the f-tree is used to determine if a union belongs to an attribute which is *leaf* in the f-tree to avoid unnecessarily recursions.

Simple Deserializer, see Algorithm 4.2, is not as simple as its counterpart but it is easy as soon as some key things are explained.
First of all, only factorization nodes of type *Union* are serialized, so we know that during the deserialization phase we only deserialize union nodes, hence the creation of a Union node just from the start of the function (*opSummation*). Then we read the children counter for this union and such many values from the input stream (note that we use *binary* format in deserializer too to match the serializer).

Now that the values for the union are read we have to use the f-tree to determine what type of factorization node each value should represent. If the current union being deserialized represents a leaf attribute (*currentAddr*) (like C, D and F in the example) we just append the values in the union node using a special *Operand* node.

If the current union represents an internal attribute node (like A, B, E) we have to check if this is a multiplication attribute, meaning that it has 2 or more child attributes in the f-tree (like A and B). If the current attribute is not product/multiplication we just add the values to the current union (*opSummation*) and as a *subtree* node we add whatever the recursion on that value will return. If the current attribute is a multiplication then we need to create a factorization node of type *Multiplication* for each value and each child of this multiplication will be the recursion result on each of the current attribute's children. For example, if the current attribute (*currentAttr*) is B it means that is has two children, attribute C and attribute D. Therefore each value of union B will have a node of type *Multiplication* that has two subtrees, one for each of the C and D attributes and their subtree nodes will be the respective recursion result.

Algorithm 4.2: Simple Deserializer

```
// @in: the input stream from which we deserialize the factorization
// @currentAttr: the current attribute node in the f-tree (initially the root)
FRepNode* dfs_load(istream *in, FTreeNode *currentAttr) {
    // we know that we only deserialize unions so create a new union
    Operation *opSummation = new Summation(currentAttr->name,
                                           currentAttr->ID,
                                           currentAttr->value_type);
    // deserialize the children count and the values for this union
    unsigned int childrenCount = read_binary(in);
    vector<Value*> values = read_binary_many(in, value_type, childrenCount);

    // now use the f-tree to infer factorization structure
    if (is_leaf_attribute(currentAttr)) {
        // just append the values to the current union and return
        for each value V in 'values' { opSummation->addChild(V, new Operand(...)); }
        return opSummation;
    } else {
```

```
        // this is an internal attribute node therefore we need to check if
        // we have to create a multiplication node for each of child values
        if (!is_multiplication_attribute(currentAttr)) {
            // not a product attribute so just store children and recurse
            for each value V in 'values' {
                opSummation->addChild(V, dfs_load(in, currentAttr->firstChild));
            }
            return opSummation;
        } else {
            // each value of the union is a multiplication operation
            for each value V in 'values' {
                Operation *opMult = new Multiplication();
                opSummation->addChild(V, opMult);
                // recurse on each attribute child and add it to this multiplication
                for each child attribute CA in currentAttr->children {
                    opMult->addChild(new Value(CA->attributeID), dfs_load(in, CA));
                } // end for each attribute in the product
            }
            return opSummation;
        } // end of if product attribute
    } // end of if leaf_attribute
}
```

4.3.5 Byte (De)Serializer

Byte (De)Serializer is an extension of the *Simple Raw (De)Serializer* technique where the only difference is that it just stores required bytes only for each value and not all the number of bytes of each data type.

4.3.5.1 Idea

If we really wanted each value to have only the required amount of bytes then somehow we would need to store that amount somewhere in the serialization in order to allow the deserializer to know how many bytes to read. It is easy to see that with millions of values, having a companion byte indicating the number of required bytes for each value

could be excessive. Therefore, we decided for each attribute to use the required amount of bytes to cover the maximum value occurred for that attribute. Therefore, we record different required-bytes for each attribute and we avoid the overhead of having them for each value since we just store them once as a serialization header at the very beginning.

We also apply the same logic to the union children counts, thus for each attribute we store two values, required-bytes for union children and required-bytes for union values. These two counters for each attribute are serialized in full binary format (8-bit unsigned numbers) at the beginning of the serialization. Therefore the deserializer will read these counters and then it will know exactly the amount of bytes to read for each union node.

4.3.5.2 Algorithms

Byte Serializer

The *dfs_save()* method is the same as the *Simple Serializer* with the only difference that the 2 lines writing to the output stream *a)* the children count and *b)* the actual values, use a special variant of the *write_binary()* method that accepts a third argument denoting the number of bytes to write from the given value.

However, in order to know this required-bytes for each attribute union children and values we have to do a pass over the factorization and gather statistics around the actual values. This means that *Byte Serializer* traverses the whole factorization twice, but as the experiments show it does not hurt a lot in processing time and helps a lot in space-efficiency.

We skip the *dfs_save()* method code since it is exactly the same as described above and we provide the first pass algorithm that gathers statistics about the unions

Algorithm 4.3: Byte Serializer – statistics gathering

```
// @attribute_info: used below = it is a field of the Byte Serializer class
// @node: the node to start gathering statistics (initially the factorization root)
// @fTree: the factorization tree of the representation
```

```
void dfs_statistics(Operation *op, FactorizationTree *fTree) {
   if (is_multiplication(op)) {
        // multiplication nodes children are unions so just recurse on them
        for each child union CU in op->children { dfs_statistics(CU, fTree); }
    } else {
        // check if the current attribute required-bytes need to be updated
        children_bytes = required_bytes(op->childrenCount);
        if (attribute_info[op->attributeID].required_union_bytes < children_bytes)
            attribute_info[op->attributeID].required_union_bytes = children_bytes;
        // check value bytes
        for each child value CV in op->children {
            val_bytes = required_bytes(CV);
            if (attribute_info[op->attributeID].required_value_bytes < val_bytes)
                attribute_info[op->attributeID].required_value_bytes = val_bytes;
            // recurse if this is not a leaf attribute in the f-tree
            if (!is_leaf_attribute(fTree, op->attributeID)) {
                dfs_statistics(CV, fTree);
            }
        } // end for each child value
    }
}
```

The statistics gathering procedure is pretty straight-forward. We do a DFS-traversal on the factorization and whenever we are at a union node we update the required bytes for the number of children and for the values of that specific attribute represented by that union node. The *required_bytes()* method returns the number of active value bytes starting from the LSB (least significant byte) to the MSB (most significant byte).

In the pseudocode above *attribute_info* is a field of the *Byte Serializer* class and its type is as shown below:

```
struct AttrInfo {
    uint8_t required_value_bytes;
    uint8_t required_union_bytes;
}
```

This structure represents the header for each attribute that is written before the actual factorization serialization as part of the header and each counter is an 8-bit unsigned integer (thus 2 bytes per attribute required).

Byte Deserializer

The *Byte Deserializer* is exactly the same as the *Simple Deserializer* with the only difference that the 2 lines where it reads from the input stream the number of children and the values themselves it uses a third argument to the *read_binary()* method that specifies the number of bytes to read.

Before calling the method *dfs_load()* we separately read the counters for the required-bytes needed for union children and values respectively.

4.3.6 Bit (De)Serializer

The final version of the serialization technique is *Bit Serializer*. As the name suggests it follows the same idea as the *Byte Serializer* but instead of working at byte-level, it works at bit-level. Therefore, instead of storing the minimum amount of required bytes for each union count and each value, it stores only the required *bits*.

4.3.6.1 Idea

The idea of this serialization technique came up after we tested applying state-of-the-art compression algorithms like *GZIP* and *BZIP2* upon our own *Simple* and *Byte* serializers. We saw that applying these compression algorithms reduced the output size by a constant factor ranging from 1-4x while at the same time increased the processing (serialization and deserialization) time significantly!

Although *serialization* is different than compression and should not be mixed (serialization is used for saving and loading a structure whereas compression is used to

exploit values to reduce size), in our case it was obvious that we could be more space-efficient by exploiting the data in our factorizations. We achieved similar or close enough compression on our factorizations in a fraction of the time required by *BZIP2* compression for example, which provides the best compression at the cost of slow processing.

I want to emphasize that serialization is different than compression and that this chapter aimed at serialization of data factorizations. But, the knowledge of our structure allows us to exploit certain factorization properties and at the same time be more space-efficient without increasing processing time significantly. Additionally, although we have some kind of compression, we do not have the drawback of standard compression algorithms (*GZIP, BZIP2*) that need the decompress the whole fragment first and then do any processing, since we are still able to deserialize each union separately and process it before we move to the next one.

4.3.6.2 Algorithms

The algorithms are identical to those of *Byte Serializer* and *Byte Deserializer* with the exception that instead of using the *required_bytes()* method it uses the *required_bits()* method to only write the specific bits required to the output stream.

4.3.6.3 Bit Stream

This serialization technique requires bit-level precision when reading and writing values, but as we know all system calls and existing functionality provided by the standard libraries work at byte-level precision.

Therefore, in order to provide this functionality we implemented custom input and output streams (*obitstream* and *ibitstream* classes) that are used upon the underlying standard binary byte streams and use those in the *Bit Serializer* and *Bit Deserializer*. These custom bit streams basically allow for a given value to write only certain bits of its memory representation and respectively can read a certain number of bits from an input stream and reinterpret them as a data type in memory.

Briefly an explanation how the bitstreams work. When a value is written or read we use internally an in-memory bytes buffer to write and read from certain amount of bits. Whenever the bytes available in the internal buffer are insufficient to satisfy a read operation it is refilled by reading bytes from the underlying input stream. Whenever the internal buffer fills (or at user's request) the internal buffer is flushed to the underlying output stream. Therefore, this implementation of bit streams works upon the underlying standard binary streams of C++ and use buffers to handle the required read and write operations.

In addition, an important feature that makes this serializer *great* is that in future work specialized read/write methods could be provided for certain data types (floats, doubles, strings) and further increase compression without adding processing overhead by applying compression algorithms. The current implementation of bit streams heavily uses C++ templates therefore this extension should be trivial to implement in a future project.

4.4 Final remarks

We described a serialization for f-trees and three serialization techniques for Data Factorizations. The serialization module provides helper methods inside the package *fdb::serialization* that allows a user of the library to serialize and deserialize a full factorization with its f-tree easily.

Namely the ***fdb::serialization::serialize(FRepTree*, ostream&)*** receives a data factorization, *FRepTRee*, and a reference to an output stream and serializes both f-tree and representation into the stream.
Its counterpart function ***fdb::serialization::deserialize(istream&)*** deserializes from the input stream and returns an *FRepTRee*.

4.4.1 Serializations illustrated

In this section we provide an illustration of the aforementioned serialization techniques and how they compare against the binary flat table serialization. The example factorization is used, see Figure 4.3.

The separator | is just used for illustration purposes to separate the different fragments of each serialization. In real-world it does not exist and the bytes of each fragment are contiguous.

Flat tuples binary serialization

```
1 1 1 1 2 1 | 1 1 1 1 2 2 | 1 1 1 2 2 1 | 1 1 1 2 2 2 | 1 2 2 1 2 1 | 1 2 2 1 2 2 | 2 1 2 1
1 1 | 2 1 2 1 2 1 | 2 1 2 1 2 2
```

total bytes = *number of tuples* * *number of attributes* * *sizeof(int)* = 9 * 6 * 4 = **216**

Simple Serializer

```
2 a1 a2 | 2 b1 b2 | 1 c1 | 2 d1 d2 | 1 c2 | 1 d1 | 1 e1 | 2 f1 f2 | 1 b1 | 1 c2 | 1 d1 | 2
e1 e2 | 1 f1 | 2 f1 f2
```

total bytes = (*number of unions* * *sizeof(uint_32)*) + (*sizeof(int)* * *number value nodes*)
= (14 * 4) + (20 * 4) = **136 bytes**

Bit / Byte Serializer

Recall that *Byte Serializer* and *Bit Serializer* use the same serialization form as the *Simple Serializer* but store only the required amount of bytes and bits respectively for each attribute union children count and union max value.

For the example we illustrate here the *Byte Serializer* just needs **1 byte** for both the union children counts and for the max value occurred in each attribute. Therefore its total serialization size would be **34 bytes**.
Bit Serializer needs **2 bits** to represent max values and max number of children occurred in each attribute so the serialization size is reduced to 68 bits, thus requiring **9 bytes**.

We should note that both these serializers require a header that for each attribute has **2 bytes** denoting the max number of bytes/bits used in each union or value (i.e. 6

attributes * 2 bytes each in the header). Thus, the total serialization size for *Byte* and *Bit Serializers* is **46** and **21 bytes** respectively.

Chapter 5

Distributed Query Processing in FDB

In this section, we present the design and implementation of D-FDB, a distributed query engine that uses FDB [**REFERENCE**] for distributed query processing on factorized data. We describe how the system integrates the *HyperCube*[**Suciu**] algorithm for shuffling the data among workers and also describe how the system can be used for Single and Multi-round executions, where single or multi refers to the number of communication rounds before the query is completely evaluated.

5.1 Motivation

Distributed query processing has become an absolute necessity in today's DBMS systems. The reason is simple, once you cannot process your data using a single machine (data too large to fit in memory or query processing too slow) you either have to partition it and process one part at a time by storing intermediate results on disk or you do distributed processing.

Utilization of many machines has become the de-facto way to scale services to support either huge number of requests or the so-called Big Data, meaning huge amount of data to be processed. There are a lot of existing systems that offer distributed query processing; almost all the current NoSQL database systems are layered upon a distributed scalable system in order to be able to achieve the high throughput and low latencies they advertise[**F1, Couchbase, BigTable, DynamoDB**]. Therefore it is natural that FDB needs to support distribution and delegation of query processing to clusters of nodes in order to enable processing on Big Data and speed up complex queries that are too slow with single node processing.

This chapter describes D-FDB, a distributed query processing engine designed to work across a cluster of nodes, using the HyperCube algorithm to shuffle data among worker nodes and FDB query engine for query processing on each site on local data partitions.

5.2 Contributions

The contributions of this chapter are as follows:

- Implementation of the HyperCube algorithm over factorizations for data shuffling in a cluster of nodes and its integration with *Bit Serializer,* as described in Chapter 4, resulting in *Bit Serializer HyperCube* which is used during distribution.
- Design and implementation of an end-to-end distributed query engine that is able to receive a query, load the input from local storage at each site, distribute data over TCP, execute the query on received data using the existing FDB query engine, and finally gather results. Major differentiation of this system from

existing ones is that we use factorizations end-to-end. ***Factorization Input => FDB Processing on factorized data => Factorization Output***

- Different distributed execution modes, namely Single round and Multi round execution. Single round execution only shuffles and transmits data between the nodes _once_ and then execute the whole query at the same time, whereas Multi round execution splits the query into individual JOINs and repeats the Single round execution for each partial query.

D-FDB at the moment supports only conjunctive queries (i.e. JOIN queries on one or many attributes).

5.3 HyperCube on Factorizations

In this section, we introduce the HyperCube algorithm [**REFERENCE**] that previous work has shown to be great solution for data shuffling in distributed query processing. In addition, we present an algorithm that explains how HyperCube works on factorizations and finally, how we integrated it with the *Bit Serializer*, resulting in a new serializer coined *Bit Serializer HyperCube*.

5.3.1 HyperCube preliminaries

In this section, we present the theoretical background behind HyperCube algorithm which is used in our *Bit Serializer HYperCube*.

A lot of novel data management systems, especially analytics engines, operating on large-scale data nowadays are equipped with large amounts of main memory which is used during the evaluation of complex analytics queries [**Spark, F1**]. Traditional systems based on secondary storage required many disk I/O operations to load and save intermediate results, thus their main bottleneck is disk I/O, whereas for an in-memory database system that bottleneck has been replaced by the communication cost incurred

during query evaluation since large amounts of data needs to be reshuffled among the workers at the beginning of each processing round.

Our focus is on conjunctive queries, which have always been important (mostly with star-joins of a large table with other smaller feature relations). Recently data engines are required to be able to process complex queries including cyclic-queries on huge tables either for analytics or for analyzing graphs for networks.
Example of a simple cyclic query is the triangle, which does a self-join two times on a relational table. A traditional DBMS would evaluate this query by doing one join first and then another join of the initial table with the intermediate result. Recent work though by [**Ngo and Veldhuizen**] and [**Afrati and Ulmann**] presented algorithms that evaluate multi-join queries, eliminating requirement for huge intermediate results. The work by *Afrati et al.* was later extended by [**Bearne et al**] who named that algorithm **HyperCube** and proved it was optimal, but its proof was not practical in a real scenario since it assumed that we can have fractional number of servers. Last year, *Suciu et al* [**From theory to practice**] provided a refinement of the algorithm that does not depend on fractional servers, thus making it practical, and showed that for many queries it can significantly reduce the amount of data communicated during query processing.

We briefly explain the idea behind HyperCube which is used in our serializer during distributed query evaluation. HyperCube is used as the data shuffling algorithm before applying a multi-way join operator on the data received. Therefore, each worker has to receive all the data he needs to correctly evaluate the multi-way join without affecting the result and at the same time retaining the *single* communication round.

We will use an example scenario to explain the algorithm.

Assume we have a cluster of P nodes (in our example $P =$ eight nodes) and our database consists of four relations:

a) *R(A, B)*
b) *U(A, C)*
c) *T(B, D)*

d) *S(C, D)*

We want to evaluate the following conjunctive query:

$$\mathtt{Q(A,B,C)} = \mathtt{R(A,B)} \wedge \mathtt{U(A,C)} \wedge \mathtt{T(B,D)} \wedge \mathtt{S(C,D)}$$

First, we need to find *ND* factors that their product equals *P* (how to find these factors is out of this project's scope and can be found in the aforementioned work, but it suffices to say that *ND* is related to the number of join-attributes). We name these factors *pi*, with *i* ranging from 1 to *ND*, therefore we have:

$$\mathtt{P} = \mathtt{p1} * \mathtt{p2} * \ldots * \mathtt{p}ND$$

Let us use $ND = 3$ and all factors equal to two ($p1 = p2 = p3 = 2$). We say that we have three dimensions and each dimensions size of two.

Our cluster of nodes is modelled into a virtual hypercube which has *ND* dimensions and in each dimension it has the respective *pi* size. Each node represents a point in this hypercube and is identified by a vector of *ND* values, one in each dimension. In our example, our cluster is formed as the hypercube illustrated in Figure 5.1.

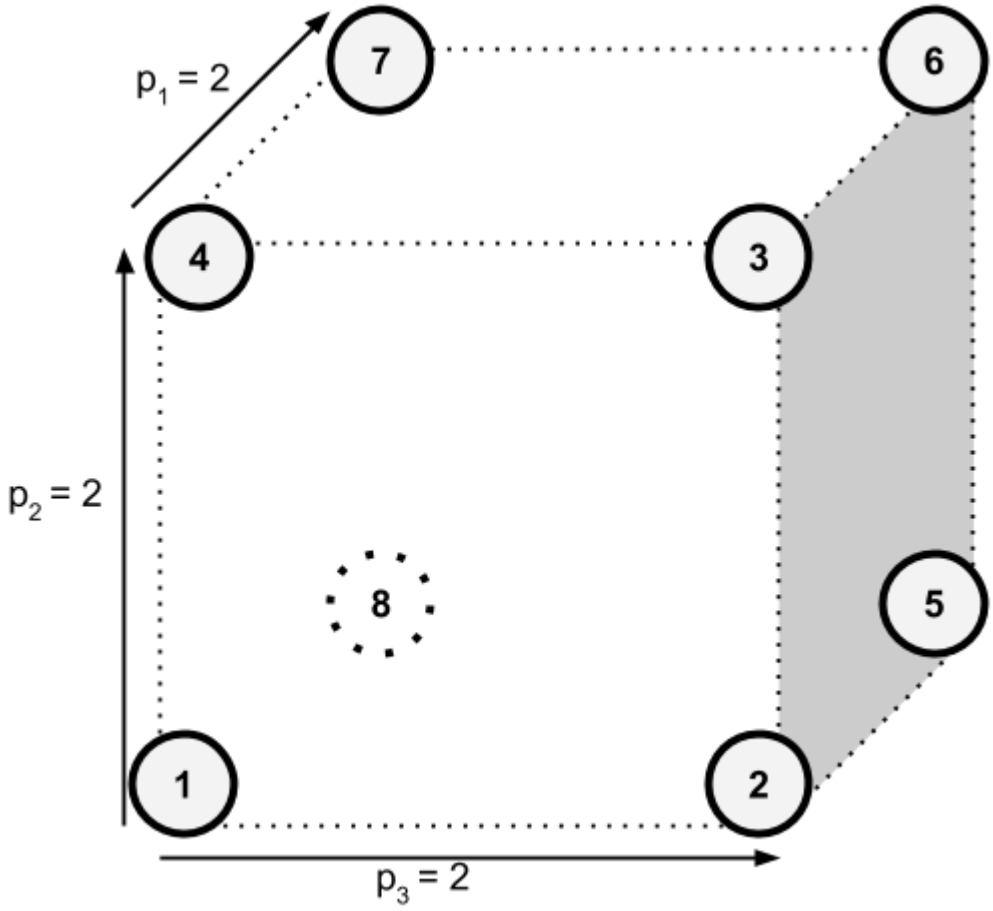


Figure 5.1: Cluster of 8 nodes in a HyperCube formation

We can see that there are two nodes in each dimension. Below we provide a possible assignment of the identifying vector for each node (multi-dimensional IDs).

Node X: [Position in *p1*, Position in *p2*, Position in *p3*]

Node 1: [0 0 0]

Node 2: [0 0 1]

Node 3: [0 1 1]

Node 4: [0 1 0]

Node 5: [1 0 1]

Node 6: [1 1 1]

Node 7: [1 1 0]

Node 8: [1 0 0]

Additionally, each dimension represents an attribute in the JOIN query. For example in our query, dimension *p1* represents attribute A, dimension *p2* represents attribute B and dimension *p3* represents attribute C. HyperCube also uses a hash function for each join/hashed attribute, chosen independently from the others, which has a co-domain of the dimension size that represents that attribute.

Furthermore, we assume that all four relations are partitioned uniformly and distributed among the nodes. Each server during the **single** communication round, will load its local partition Zi of each relation Z from its local secondary storage and for each tuple T decides which nodes should receive it as follows:

(1) Create a multi-dimensional vector ID similar to those assigned to each node, let's call it CTV, initialized with *, hence [* * *]

(2) For every attribute t in relation Z that is among the JOIN-attributes of the query, it hashes the value $T[t]$ and assigns the hashed value to the vector CTV

(3) If CTV contains no * then it can be used as the multi-dimensional ID for the node that should receive the tuple. If CTV contains * it means that the current relation Z does not contain all the hashed/join attributes, therefore that tuple T should be sent to more than one nodes.
To identify the required nodes we use CTV, with every * acting as a wildcard for ALL values in that dimension, meaning that if **CVT = [0 1 *]** the tuple should be sent to the nodes with IDs **[0 1 0]** and **[0 1 1]**.

To make this clear, we can see from our example that all tuples from relation R will be sent to two nodes since only dimensions *p1* and *p2* can be defined by its tuples. This holds for all relations in our example since all of them only contain two out of the three hashed attributes.

HyperCube's advantage over other shuffle techniques (i.e. hashing an attribute to all nodes) is that it is more resilient to data load imbalance (a.k.a skew) since it is more

difficult to send the same value for a column to the same node since it depends on the other hashed columns too.

The intuition to HyperCube's correctness is that since we use the same hash function to hash values of the same attribute/column, then all required tuples to evaluate correctly the JOIN will end up in the same node. The *wildcard* is used to ensure that even if a tuple does not contain a hashed attribute it will be sent to the nodes for the missing dimension since they might need that tuple based on the rest values which might contain a hashed attribute.

The methodology to identify the proper values for *ND* number of factors/dimensions and the size of each dimension are not part of this project, thus not presented here. In experiments, we used the query configuration files to specify the HyperCube dimensions required for the query we wanted to evaluate.

5.3.2 Bit Serializer HyperCube

Bit Serializer HyperCube's main purpose is to be used during the communication stages in D-FDB query processing phase. Each node needs to send data over the wire to other nodes, therefore we use this special serializer to take into account the HyperCube configuration used and serialize only the fraction of the factorization required.

5.3.2.1 Arguments

Before explaining the arguments of the algorithm we explain some important topics used throughout this chapter, using an example f-tree, see Figure 5.2.

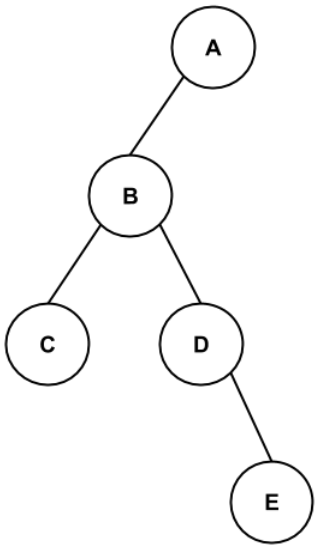


Figure 5.2: Example f-tree

This f-tree has five attributes, and each attribute internally gets an ID ranging from zero to (*N-1*), where *N* in this case equals five. Assume that the IDs for these attributes are as below:

$$\mathbf{ID(A) = 0}$$
$$\mathbf{ID(B) = 1}$$
$$\mathbf{ID(C) = 2}$$
$$\mathbf{ID(D) = 3}$$
$$\mathbf{ID(E) = 4}$$

Moreover, let us consider that we want to use HyperCube and hash on attributes A and E (we support hashing attributes regardless their position in the f-tree). Also, our cluster will contain 6 nodes. There are four possible HyperCube configurations in order to use all nodes, as shown below.
Notation K x M means that we assign a dimension of size K to attribute A and a dimension of size M to attribute E.

Conf 1: 1 x 6
Conf 2: 6 x 1
Conf 3: 2 x 3
Conf 4: 3 x 2

In addition, each node will be assigned a *multi-dimensional ID* based on the HyperCube configuration used. For this example, let's use the third HyperCube configuration (*Conf 3*), thus creating the node IDs below (basically we iterate over all possible values in each dimension).

Node 1: [0, 0]
Node 2: [0, 1]
Node 3: [0, 2]
Node 4: [1, 0]
Node 5: [1, 1]
Node 6: [1, 2]

Bit Serializer HyperCube was designed to accept the following arguments:

(1) The factorization to be serialized
(2) Bitset or vector (array) with size the number of attributes in the f-tree, where each *set* bit corresponds to an attribute that is to be hashed
(3) Vector (array) with size the number of attributes in the f-tree. Each value in this array corresponds to the node's dimension ID for that attribute taken from its multi-dimensional ID

Let us provide the actual arguments used by *Bit Serializer HyperCube* for our example.

(1) The factorization to be serialized
(2) Bitset of size six with the bits *set* for attributes A and E: [**1 0 0 0 1**]
(3) For each node we call the serialize method of the serializer passing in the multi-dimensional node ID expanded to have size of N.

For example, if we were to serialize for node 6 the vector ID [**1 0 0 0 2**] would be used whereas for node 2 [**0 0 0 0 1**].

As you can see each node's expanded ID is a vector of size N (number of attributes). Each position *T* in this vector either has zero if attribute with ID T is *NOT* among the hashed attributes or has the node's ID in dimension T as specified in the node's multi-dimensional ID.

5.3.2.2 Hashing and HC_Params

HyperCube's performance depends on value hashing and proper use of hash functions. In this project we decided to use the same hash functions as existing work that showed good results [**Suciu**]. The hashing library used is *MurmurHash3* which is open-source and available online. The library provides methods that given a series of bytes create hash values of size 128-bits and 32-bits. We decided to use the 128-bit version and just use the first 64-bits (starting from the Least-Significant-Bit).

Additionally, these hash functions accept a *seed index* as argument which affects the result hash values. In order for the HyperCube to work as expected we need to use the **same** seed index for attributes that are to be joined together, or attributes that are named differently in the factorization but represent the same logical attribute. In addition, it would be better to use different seed indices for different joined attributes to avoid distribution issues that might result in skewing of data partitioning and shuffling. Therefore, we have a pool of some seed indices that are given to each hashed attribute (seeds taken from online prime number resources and previous work on HyperCube).

HC_Params

The structure *hc_params* is a structure passed as argument in the serialize function of *Bit Serializer HyperCube* and contains the three arguments described in the previous Section 5.3.2.1 to allow the serializer to use the right hash function for each attribute value during validity check.

5.3.2.3 Algorithms

In this section we present the algorithms behind bit serialization using HyperCube.

Recall, that HyperCube shuffling hashes the value in each *hashed attribute* and based on these hashed values sends the whole tuple to the nodes that have their multi-dimensional IDs matching the hashed values, attribute-wise (the hashed value of a column has to match the node's dimension ID on that column).

HyperCube implementation in flat databases handles tuple as a whole, therefore can apply hash functions in all the required attributes and find the matching nodes for the tuple instantly. In our case, factorizations, do not have all the information about a tuple in a single place since each tuple is assembled by retrieving a value from each attribute union along the factorization.

A naive approach would traverse the factorization, hash the values in each union and send them to each node that matches the hashed value in its multi-dimensional ID. This would lead to incorrect results since a single attribute union cannot determine the destination nodes. In order to be able to decide whether each value in a factorization should be sent to each node we have to make sure that the value exists in at least one tuple that is valid for that node, and we cannot know that before traversing all attribute unions in the factorization.

As a result, our algorithm consists of two phases, namely the ***masking phase*** and the ***serialization phase***. During the *masking phase* we create bitset masks for each union denoting whether each value is valid to be sent to the examined node, and during the *serialization phase* the valid values are serialized in the exact way *Bit Serializer* works.

An important optimization in the algorithm is that during the second phase there is no need to visit unions that do not have any valid values to be serialized. Therefore, there is lot of gain since we can skip complete subtree branches from the top-most point we notice that a value is invalid.

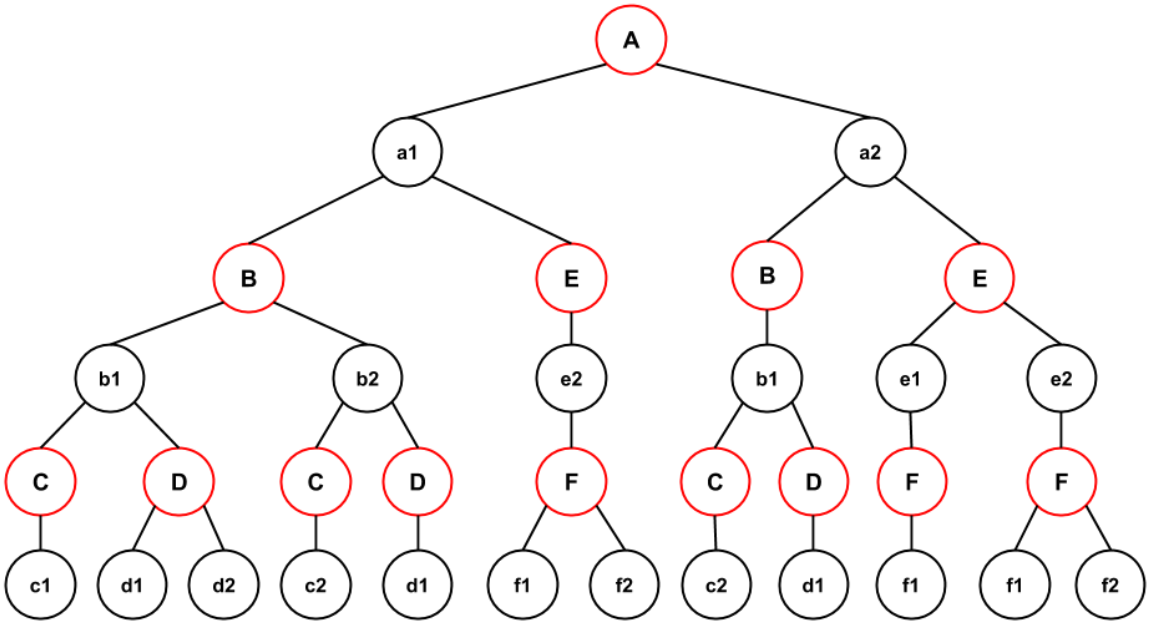


Figure 5.3: Example factorization

For example, assume we are serializing the factorization seen in Figure 5.3 and the node we examine has the multi-dimensional ID [**1 0 0 0 1 0**]. If the value **a1** hashes into zero (0) then we know that the whole branch under *a1* should not be visited since it will be not serialized for this node (dimension ID for this node is one - 1).

It is very important to distinguish between deciding if a value *is valid* for a node and when it is not. When we are at a union examining the values to be serialized we can reject a value instantly if it does not hash to the proper value to match the node's multi-dimensional ID, thus complete subtrees, but we cannot know for sure if it is valid unless we examine its entire subtree. For example, if **a1** hashes into one (1), which is valid, it might still be invalid for the node we examine if **e2**, which is the other hashed attribute in our example case, does not hash into this node's dimension ID.

In order to take advantage of the opportunity to skip subtrees we have to keep-track of which values are valid in each union. One way was to create a virtual layer upon the factorization that keep this information, but we decided to simply use a vector to hold all union states (value bitmasks). Once the *masking phase* is finished the states inside

the vector should be in the order we are going to visit the valid unions during the serialization phase, which is not too difficult to maintain since we are doing a DFS traversal in both cases.

In the rest of this section we will provide and explain the algorithms for the two phases that implement the *Bit Serializer HyperCube*. As can be seen from the code, we included the masking phase into the first round of *Bit Serializer* that gathers statistics about maximum values and bits required, therefore we still only do two passes over the factorization.

Masking phase - statistics gathering - first pass

Algorithm 5.1: Bit Serializer HyperCube – gather statistics – first pass

```
// @node: a node in the factorization to start serialization (initially root)
// @fTree: the f-tree used by the current factorization
// @hc_p: the HyperCube parameters as defined in Section 5.3.2.2
// @return: True iff *node contains values to be serialized
bool dfs_statistics(Operation *op, FactorizationTree *fTree, hc_params *hc_p) {
    if (is_multiplication(op)) {
        if (op->children is empty) return false;
        bool valid_child = true;
        for each child attribute CA in op->children {
            // recurse on each union and make sure all of them are valid
            valid_child &= dfs_statistics(CA, fTree, hc_p);
            if (!valid_child) return false;
        }
        return true;
    } else if (is_union(op)){
        // special treatment for unions since they contain the values to be hashed
        return handle_union(op);
    }
}
```

Algorithm 5.2: Bit Serializer HyperCube – gather statistics – first pass: handle union

```
// @mMasks: class field - vector that contains the bool masks for each union
//
// @op: the union node in the factorization to gather statistics
// @fTree: the f-tree used by the current factorization
// @hc_p: the HyperCube parameters as defined in Section 5.3.2.2
// @return: True iff *op contains values to be serialized
bool handle_union(FRepNode *node, FactorizationTree *fTree, hc_params *hc_p) {
    // add our bitmask into the states vector keep reference to our state's position
    mMasks.push_back(); iMask = mMasks.size() - 1;

    // now we check each value if it is valid and if yes make sure
    // that its subtree has a valid result too before masking it valid
    for each value child CV in op->children {
        // make sure that the value hashes to the right Node dimension ID
        // if this union is of a hashed-attribute otherwise the value is
        // always valid and will be serialized
        if (is_valid_value(CV, hc_p)) {
            if (is_leaf_attribute(op->attributeID, fTree)) {
                // leaf attribute means valid value instantly
                mMasks[iMask].push_back(true);
                // also gather statistics about required bits
                update_required_bits(CV);
            } else {
                // this is not a leaf union so we have to make sure
                // that the subtree contains valid values too
                if (dfs_statistics(CV, fTree, hc_p)) {
                    // finally valid value
                    mMasks[iMask].push_back(true);
                    update_required_bits(CV);
                } else {
                    // the value's subtree is invalid so the value is too
                    mMasks[iMask].push_back(false);
                }
            }
        } else {
            mMasks[iMask].push_back(false);
        }
```

```
    } // end for each value child

    // count the valid children
    valid_children = count(mMasks[iMask], true);

    // make sure that we have valid values to serialize otherwise
    // we have to return false such that our parent knows we are invalid
    if (valid_children == 0) {
        // remove our state from the masks vector and all of our descentants
        mMasks.resize(iMask);
        return false;
    }

    update_required_union_bits(valid_children);
    return true;
}
```

Entry point of the *masking phase* is method *dfs_statistics()* which is called initially with the root of the factorization and recursively visits all other nodes. As previously explained, the algorithm cannot determine if a node or value is valid without recursing on its subtree. We also differentiate the two scenarios, *a)* Multiplication nodes and *b)* Union nodes in the factorization. When the current node is multiplication we want to make sure that we have valid values in **ALL** the subtrees since a tuple is assembled by the product of these subtrees, thus one of them being empty means no result. If the current node is a union it is treated separately by the *handle_union()* method, see Algorithm 5.2.

Let's delve into the union handling. First of all, we create a union state and put it in the masks vector, at the same time recording its index position. The reason we want to use an index and we don't always refer to the top state is that we will recurse again, so possibly another state will be pushed (by a descendant), hence we need a way to access the state for the current union.

The logic behind HyperCube is in the next lines where we iterate over all the current union's values (**Lines XX**). For each value CV we first check if it is a valid value. The validity of a value depends on whether this is a union of a hash-attribute or not. If it isn't, then the value is automatically valid, otherwise we hash the value and check it against the multi-dimensional ID of the node (use of the *hc_params*). If the value is not valid then we append into our mask-state a *false* bit and continue to the next child immediately, thus skipping the invalid subtree entirely. If the value itself is valid, we have to make sure that its subtree is valid before marking it as *true*, therefore if there is a subtree (not leaf attribute) we recurse and only when the returned result is success we mask the current value as valid.

Whenever the value is valid, we also calculate and update the required bits for that attribute. When all the children have been processed we have to ensure that this union has at least one valid value, otherwise it should not be serialized. If it does not, then we return false immediately denoting this union invalid.

It is very important to understand the reason why the states vector is being resized to match the current union's index (*iMask*). Note, that resizing the vector to size N, all elements above and including N will be removed. Observing the traversal over the values and the recursion calls it is easy to see that we do a DFS-like traversal over the factorization. However, not all nodes will be visited since at any point one node might be invalid and therefore instantly return false to its parent, hence propagating the failure upwards to the current union. Therefore the number of states pushed into the masks vector is undetermined and can range from *zero* to the *number of nodes in the subtree* of each value. When the current union is invalid (children list empty) it means that none of our valid children (if any) will be serialized in the result, therefore their states need to be removed from the vector.

During the serialization process we do a DFS traversal on the factorization and each time a union node is encountered we get the first available state from the vector and serialize recursively only the valid values. Therefore, the states vector should only contain masks for the unions that are valid to be serialized and in the order they will be serialized.

To complete the union handling, the function just updates the required bits for the union children counter in case it is valid and return success to its caller.

Serialization phase - second pass

The second phase, the serialization, of our algorithm is identical to the second phase of the regular *Bit Serializer*. The only difference is that instead of serializing all the values in each union we use its mask state to identify the valid children and serialize those only. Respectively, recurse on them only. Each union takes the next available mask state from the states vector, starting from index 0 moving upwards.

Complexity

The complexity of serializing a factorization using *Bit Serializer HyperCube* and *Bit Serializer* is asymptotically the same. They both incorporate two passes over the whole factorization.

The HyperCube version, however, has the additional overhead of hashing the values in unions of hashed attributes. Although it has constant overhead it still adds up to the total processing time. Furthermore, this version might skip certain subtrees when values are invalid which can speed up the second phase significantly. Both points affect runtime of the HyperCube serialization but its performance strictly depends on the values of each factorization and on the number of hashed attributes.

Deserialization

Bit Serializer HyperCube is perfectly compatible with the regular *Bit Deserializer*. Therefore one can use it to deserialize hypercube serializations into factorizations.

5.4 System Architecture

In this section, we present the architecture of *D-FDB* and how its individual components communicate and coordinate during the execution of a query. An abstract overview of the whole distributed architecture is illustrated in Figure 5.4.

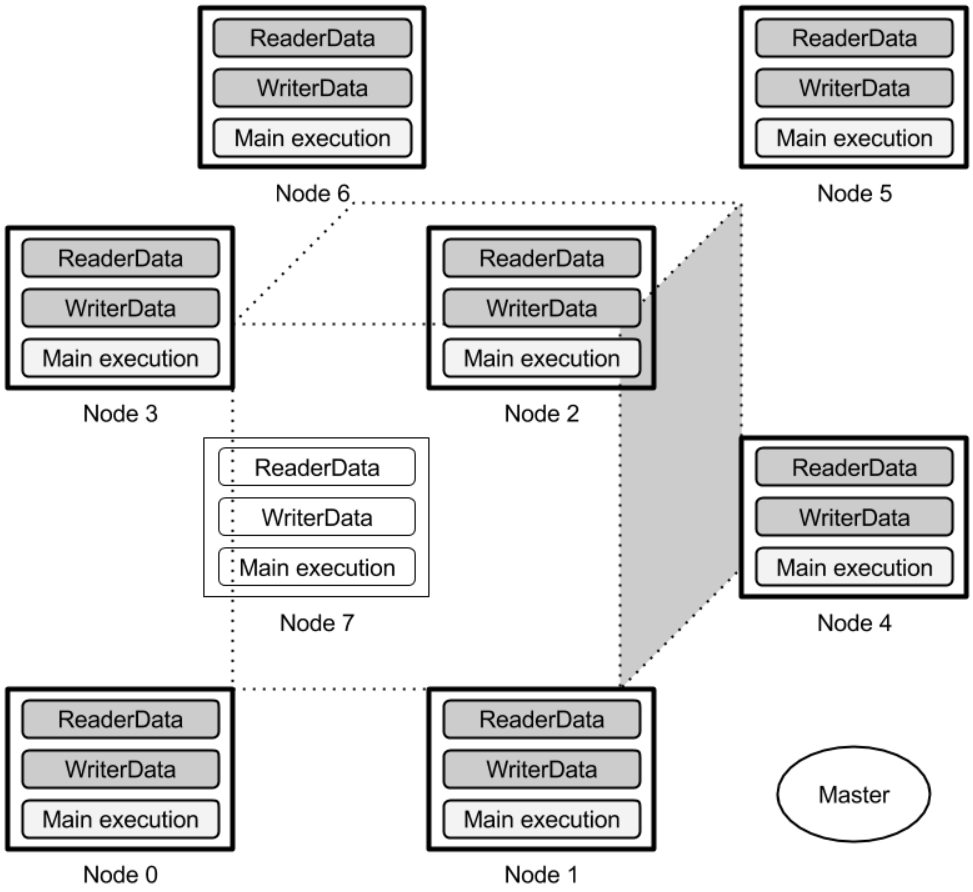


Figure 5.4: D-FDB Architecture – cluster of 8 nodes in a 3-D HyperCube formation

5.4.1 Architecture model

The design of the system follows the *master-worker* model, where one node acts as a master which coordinates the worker nodes. The worker nodes form a cluster and its size can range from one to multiple nodes. In D-FDB implementation the master node is used only for coordination, to provide abstract synchronization during the different stages of execution to the worker nodes who are working entirely asynchronous while a query is being processed. Therefore the communication with the worker nodes is kept minimal.

Additionally, we do not require the master node to be a standalone machine, hence giving someone the option to have any of the worker nodes in the cluster act like the master node of the system too. The master's responsibility is just to receive a single message from each worker node at the beginning of each phase denoting their status (ready to proceed) and responding back with the next phase initiation signal.

Moreover, the entire distributed query engine can be used entirely on a single machine using separate processes to simulate the nodes of the cluster. This not only provides an easy way to debug and test the framework in its current and future state but it also lets us exploit multi-processor and multi-core machines (machine resources are *shared*). The current FDB implementation is centralized and single threaded so this customization is very welcome since it enables parallel query execution.

All configuration options about the topology (master, worker nodes) and the query to be processed are specified using two simple configuration files, see Section 5.5.

5.4.2 System Protocol

The system, once given a distributed query processing request, starts the **distributed runner** in each site (node). This is a special class that determines whether the running process is a master or a worker and initiates the main execution thread.

The distributed execution of a query has the following four stages.

(1) Initial Handshake

The purpose of this stage is to ensure that all nodes are up and running, ready to process the query. Each worker node sends a *hello* message to the master in order to signal its existence in the cluster. Once the worker sends this message, it blocks until the response from the master comes back. Before sending the *hello* message though, each worker spawns a separate thread/process and runs the **ReaderData** (see Section 5.4.3).

The master node on the other side, upon starting, waits to receive N *hello* messages, where N is the number of worker nodes. As soon as all have been received, it

broadcasts to the workers a message signaling initiation of the next stage, *Connection Establishment*.

(2) **Connection establishment**

In this stage each worker node establishes TCP connection with all other workers, in order to be able to send and receive messages without establishing new connections every single time during the query execution.

When a node receives the *Connection Establishment* initiation message, it spawns a new thread/process and runs the **WriterData** (see Section 5.4.3) which is going to initiate connection to *ReaderData* of all the worker nodes in the cluster. When all the connections have been established and cached, each worker sends a *ConnectionEstablishmentFinished* message to the master and blocks.

The master again just waits to receive N *ConnectionEstablishmentFinished* messages and once it does, it broadcasts the *QueryExecution* message to signal initiation of the next stage.

(3) **Query execution**

This is the most important stage of the whole distributed query processing. In this stage each worker will parse the query configuration, load the local inputs in memory and will evaluate the query based on the distribution mode requested (Single round vs Multi round). When the query has been fully evaluated, the *QueryExecutionFinished* message is sent to the master.

The master waits for N *QueryExecutionFinished* messages and once all have been received, it broadcasts the initiation message for the final stage.

(4) Results gathering

This is the stage where information regarding the partial results on each worker is being communicated. For example, each worker can send a path to the master node where the partial query result is located. Moreover, this stage is the final stage of the distributed query processing so the workers can do any cleanup on resources allocated and terminate gracefully.

The whole query processing is done in stage 3, including data partitioning, shuffling and f-plan execution on local factorizations. The rest stages were required for bootstrapping the system and in a real-world scenario where a cluster of nodes is already up and running they would not even exist. Therefore, it is nice to have the query processing isolated in order to be able to reason correctly about each processing phase and also about the end-to-end experience of the system which can easily be measured on the master node.

5.4.3 Communication in the cluster

In the previous section, *System protocol,* we mentioned **ReaderData** and **WriterData**. These two classes, *runnables in separate threads*, are responsible for all the data transmission among worker nodes, both in Single and in Multi round execution.

ReaderData is initiated during *Stage 1* and it starts by creating a TCP socket listener ready to accept connections from the *WriterData* threads during *Stage 2*. *WriterData* is initiated during *Stage 2* and instantly tries to connect to all worker nodes, specifically to the *ReaderData* threads running on them.

In *Stage 3*, query execution stage, the two *services* are responsible to send and receive factorizations over the network using TCP. Specifically, *ReaderData* in each execution round (one for Single-mode, many for Multi-mode) waits on each of the N worker TCP streams in order to receive this round's factorization from that node. It uses the *Bit Deserializer* to deserialize data received into in-memory factorizations, skipping all those that are empty or invalid. On the other side, *WriterData* is responsible to take the input factorizations (or the previous round's result) and serialize it to all worker nodes using the *Bit Serializer HyperCube* which implements the shuffling HyperCube algorithm on factorizations. If no values are valid to be sent to a node, the empty factorization is sent.

Some design rules were applied for data communication and are enumerated below.

(1) Each worker will send a single factorization to each other worker during each communication round of the query execution stage. In case no valid values exist then the empty factorization is sent (the f-tree and a zero-sized factorization).
(2) All factorizations sent in the same communication round should have the *same f-tree*, because they are merged into a global one which is then used for the f-plan evaluation (query operations). This is due to a limitation of FDB being unable to merge factorizations of different f-trees.
(3) The factorizations are serialized into the TCP streams using *Bit Serializer HyperCube* and deserialized on the other side using *Bit Deserializer*.

The decoupling of data communication from the execution thread, even reading data from writing data, turned out to be very useful because it made the implementation more modularized and more extensible, thus giving us the opportunity to make improvements on any side without affecting the other. Most importantly, since reading and writing are decoupled and they run in parallel, we achieve concurrency throughout the communication phase since a worker can send data to one node and receive from another at the same time.

5.4.3.1 Ordered communication or not

In all distributed systems there is a certain point of time where each node has to communicate data with other nodes. Many times all nodes have to send data to many other nodes, see Figure 5.5.

One issue we faced while designing the system was the actual order of data transmission among worker nodes. It is a problem that appears in every distributed system but there is no published work on how this should be done, at least reasoning about efficiency. Even related work studying algorithms for better data partitioning and shuffling, like HyperCube, very relevant to this problem, do not address this decision-problem in their publications. We had numerous ideas and thoughts but we concluded in two designs and finally implemented one, with the other being added for investigation in the future.

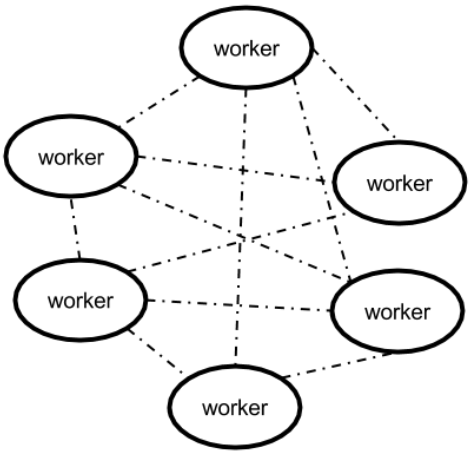


Figure 5.5: Worker nodes communication data

We now present the two ideas considered for communication inside the cluster.

Idea 1 - Ordered communication

In this approach, *ReaderData* and *WriterData* will read and write data to other workers following a specific order aiming to achieve good throughput. ReaderData fully reads a message from one node before proceeding to the next one, while WriterData fully writes data to one node before writing to the next one.

The specific ordering in reads and writes aims to maximize possibility for high throughput and maximum overlap between reads and writes in the cluster. We want to avoid having a node waiting to receive data from one node, while another node is blocked waiting to send data to a third busy node. For example, let us go through the following communication ordering in a cluster of 5 worker nodes.

Writing ordering	**Reading ordering**
Node 1: 2 3 4 5	*Node 1: 5 4 3 2*
Node 2: 3 4 5 1	*Node 2: 1 5 4 3*
Node 3: 4 5 1 2	*Node 3: 2 1 5 4*
Node 4: 5 1 2 3	*Node 4: 3 2 1 5*
Node 5: 1 2 3 4	*Node 5: 4 3 2 1*

The first block defines the order of writes for each node. Node 1 for instance, will send data to node 2, then node 3, then node 4 finally node 5. The second block defines the order of reads for each node. For example, Node 3 will read from node 2, then node 1, then node 5 and finally node 4.

The important thing in this ordered communication, is that if we distinguish 4 communication rounds (vertical division) we can see that no node is ever stalled or blocked without reading or writing. Additionally, there is a pairing between the reads and the writes, which means that for any given node A, the writes *targeting* A will be done in the same order as A will do the corresponding reads. In our tests, the idea works as expected, however, there were some cases when a node had to write more data than the rest, thus causing some nodes to wait for it to finish.

Idea 2 - Round robin communication

A second approach to the communication problem utilizes round-robin communication. The intuition is that instead of fully communicating with one node before moving on to the next one (either when writing or reading), we could write less to all nodes and iterate more times.

For example if node A has to send data of 1000 MB to all other nodes, instead of sending the whole 1000MB at one node at a time, it sends the first 100MB to all of them, then the next 100MB to all of them, and so long. This approach aims to keep all nodes busy at all times and possibly avoids having nodes blocked at a node that is doing a long read/write. For each partial data communication we can either use ordering like above or random pairing.

The disadvantage of this approach compared to the ordered one is that we need to keep track of the required information while sending (or receiving) for all nodes. For example, when serializing a factorization (*Bit Serializer HyperCube*) we use some statistics and a vector that holds a state for each union node. In this approach we have to calculate and have in-memory this information regarding all nodes.

Unfortunately, due to limited time we did not compare the two solutions and only implemented the first one.

5.5 Query processing and configuration files

In this section we will provide a description of the two modes supported for distributed query execution, namely Single and Multi-round execution, and then explain how these are implemented during the *execution stage* of our system. Finally, we present the configuration files we use to specify the type of execution and the query to evaluate.

5.5.1 Single vs Multi round

The distributed query engine presented in this chapter supports two modes of execution, with each mode being different in the number of communication (hence computation) rounds before completely evaluating the query.

5.5.1.1 Single round

Single round execution refers to multi-way JOIN operations when a query requires joining more than one attribute. Single round execution is based on the use of the HyperCube algorithm which partitions and shuffles data considering all the attributes to be joined at the same time.

A lot of work has been done by researchers investigating the costs in a distributed system and it is widely acceptable that a major bottleneck in distributed execution is the communication among worker nodes. Therefore, people try to devise algorithms that try to eliminate this cost as much as possible, hence HyperCube and its variations, whose aim is to reduce communication rounds to a minimum.

To summarize, single round execution means that each worker will send data to other workers only once, hence the single communication round, followed by the complete evaluation of the multi-attribute JOIN producing the final result.

5.5.1.2 Multi round

Multi round execution refers to the evaluation of complex queries by partially processing the query in each round until the whole query has been processed. For example, if the query requires joining on 3 attributes we could either have an execution that evaluates the two joins in the first round and the last one in the third round or we could even have three rounds evaluating a single join in each round.

With multi round execution one can investigate more complex topics like query decompositions into smaller queries that are easier to evaluate in separate rounds, that at the same time do not necessarily need to be single attribute JOIN operations.

However, D-FDB at the moment only supports conjunctive queries (JOIN operations) and a multi-round execution evaluates one join at a time.

5.5.2 Query execution phase

In this section we will describe how the above two modes are implemented in the proposed distributed system, what are their limitations and how can they improved.

Query processing and result evaluation corresponds to *Stage 3* as described in the *System protocol*. Details about the query and the cluster topology are specified in configuration files (will be described in the next section).

First, we present the steps taken to process a single round evaluation and then explain how D-FDB builds on-top of that to provide multi round execution. Each worker has three threads running, as previously described, *a)* main execution thread, *b)* ReaderData for receiving data and *c)* WriterData for sending data.

(1) Execution thread starts loading the input factorization in memory (using the configuration files to locate them).

(2) Then it signals ReaderData to start accepting factorizations from other workers while at the same time the input factorization is given to WriterData in order to begin writing data to other workers.

(3) Once all factorizations from other workers have been received, the empty ones are dropped and the valid ones are passed over to the main execution thread. Then the special *merge_same* operator is used to merge the partial factorizations received into a single factorization (recall that all factorizations sent in the same round use the same f-tree).

(4) Last step of the processing uses the *f-plan executor* in order to evaluate the query specified in the query configuration file on the local factorization and produce the factorization result.

(5) Finally, each worker notifies master node that the query has been completely evaluated.

The multi round execution is similar, wrapping steps 1 to 4 into a loop, one iteration for each round, where the input for the first round is loaded from local storage and in consecutive rounds we use the result of the previous round as input to the next round.

Merge Factorizations

In this section we provide the **merge_same** operator that given a list of factorizations of the same f-tree merges them together into a single factorization.

// **TODO if there is time!!!!**

Mention the **merge_under_groot** ...

5.5.3 Configuration files

In this section we introduce the two main configuration files used that specify the network topology and the query to be evaluated.

5.5.3.1 Distributed Settings configuration

Distributed Settings configuration file

```
##############################################################################
###### LINES STARTING WITH '#' or whitespace are skipped during parse!
##############################################################################

# number of nodes in the network
4
# all the IPs for each worker node (IP or IP:PORT format supported)
# each node will also receive an identification ID (uint32_t) based on its order
xxx.1.xxx.36:11110
xxx.1.xxx.39:11110
xxx.1.xxx.60:11110
xxx.1.xxx.65:11110

# specify the master node (IP or IP:PORT format supported)
xxx.1.xxx.36:11100

# now the query path
/home/lambros/dist-execution/dist_query.conf

# order of communication - data distribution
# we follow a cyclic (shifting policy)

# WRITING ORDER - we will have N lines, one line for each node that defines
# how that node should send its data
1 2 3
2 3 0
3 0 1
0 1 2
# READING ORDER - we will have N lines, one line for each node that defines
# how that node should read data
3 2 1
0 3 2
1 0 3
```

```
2 1 0
```

The excerpt above is part of a sample settings configuration file that contains all the required information about the network topology regarding our cluster of nodes. It starts by mentioning the number of worker nodes in the cluster, followed by their IPs. Recall that you can use different ports on the same machine (same IP) to simulate different nodes. In the next line (not commented) we specify the master node. The master node is *required* to have a different port than ***ALL*** worker nodes but is *not required* to have different IP, thus being on different machine.

A path to the query configuration to be evaluated is provided in the next line and finally we have the communication ordering as explained in Section 5.4. For the writing order we have one line for each node that defines the order in which the node should read data from, and for the reading order there is a line for each node that defines the order in which the reads should be made.

In this simple example the ordering could be determined dynamically in runtime since it follows a pattern (cyclic shifting), but we decided to keep it in the settings file in case we want to try different orderings in the experimental evaluation, thus avoiding source code changes.

5.5.3.2 Distributed Query configuration file

In this section we present an example query configuration file.

Distributed Settings configuration file

```
# number of input factorizations
1
# all the paths to the factorizations (serializations) to be used as input
# ---- NOTE::: if the node does not find the path specified it will try to
#           load the path suffixed with '-n-NODE_ID-' before the extension
/home/lambros/dist-datasets/somedataset/nodes_4/input-groot.dat
```

```
# F-PLANS follow for each of the inputs to be applied during loading.
# each F-PLAN has a single line of an integer N, followed by N lines with actions
0

# query F-PLANs to be applied on the local factorization in each round
# number of plans (one plan for each communication round)
1

2
merge attr_1 attr_2
merge attr_1 attr_3
end

# HyperCube configuration
# a single line will contain all the attribute names of the combined result
# which will give each attribute a GLOBAL ID (their position in the line)
# that is used internally to map the attributes from local representations.
attr_1,attr_2,attr_3,attr_4,attr_5,__g_root_

# the hashed columns should be specified now starting from 0 to N using the
# GLOBAL IDs (position in the above line) (separated by space)
# (if there are more than 1 attributes that correspond to the same attribute
# logically but have different names just use one of them and group them below)
2 1
# the dimension for each hashed attribute (separated by space)
2 2
# group synonym attributes that have different names but are the same
# i.e. id_0, id_1, id_2 which can be renamed to avoid the limitation of FDB
# to handle same name attributes
# one number N specifies the number of groups and then for each group one line
# - each group should start with the attribute that was specified in the
# - hashed columns above!
1
attr_3,attr_1
```

We start by defining the path to the input factorization that will be loaded by each worker from its local secondary storage. A feature that we found useful to have is that

if the path specified does not exist or fails to open, then each worker will try to load the path suffixed by its numeric ID.

For example if the worker on node 2 tried to open the path on the above configuration file and failed, it would then try to open the following path this time (note the *-2* suffix): **/home/lambros/dist-datasets/somedataset/nodes_4/input-groot-2.dat**

This little feature allowed us to have the input factorizations for all worker nodes in the same common directory and each node would end up loading the correct factorization, otherwise we would have to separately ship data to each node initially or use different configuration files at each worker.

The following group of options specifies an f-plan to be applied on the input right after loading it in memory, and consists of one line denoting how many lines followed with the f-plan operations.

In addition, we continue to the f-plans of the main query. The next line denotes the number of f-plans we want to evaluate. When this value is 1 we simulate Single round execution whereas when this value is more than 1 we simulate Multi round execution with each of the following f-plans be applied at the corresponding round.

The rest of the configuration is related to the HyperCube configuration. We start by enumerating all the attributes that exist in the factorizations (they don't have to be in any specific order and their order does not relate to the IDs given to them inside their respective factorization f-trees).

Right below the attribute names, we specify the attributes to be hashed. Each hashed attribute is specified by using its position in the line above where all the attributes are enumerated. In the example above the hashed attributes are *attr_3* and *attr_2*, hence the IDs 2 and 1(indexing starts from 0). Moreover, the implementation of the Multi round execution uses one hashed attribute at each round, but can easily be extended to support arbitrary f-plan by adding more options to the configuration file.

Below the hashed attributes, we specify the dimension size for each one of them in the hypercube (in the above example both attributes got a dimension size of two, thus requiring four nodes). Each node will automatically get a multi-dimensional ID based on these dimensions.

The last part of the configuration makes it easy for us to group different attribute names that refer to the same attribute. We wanted this functionality since in HyperCube we want to use the same hash function for a specific hashed attribute, therefore we had to somehow group all the attribute names referring to *attr_3* together and assign them the same hash function (seed index). Additionally, current FDB implementation has the limitation that in an f-tree we cannot have an attribute name more than once, so many times we had to rename some attributes and this configuration helped us overcome this limitation easily.

This concludes our configuration files regarding query execution.

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 – 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.

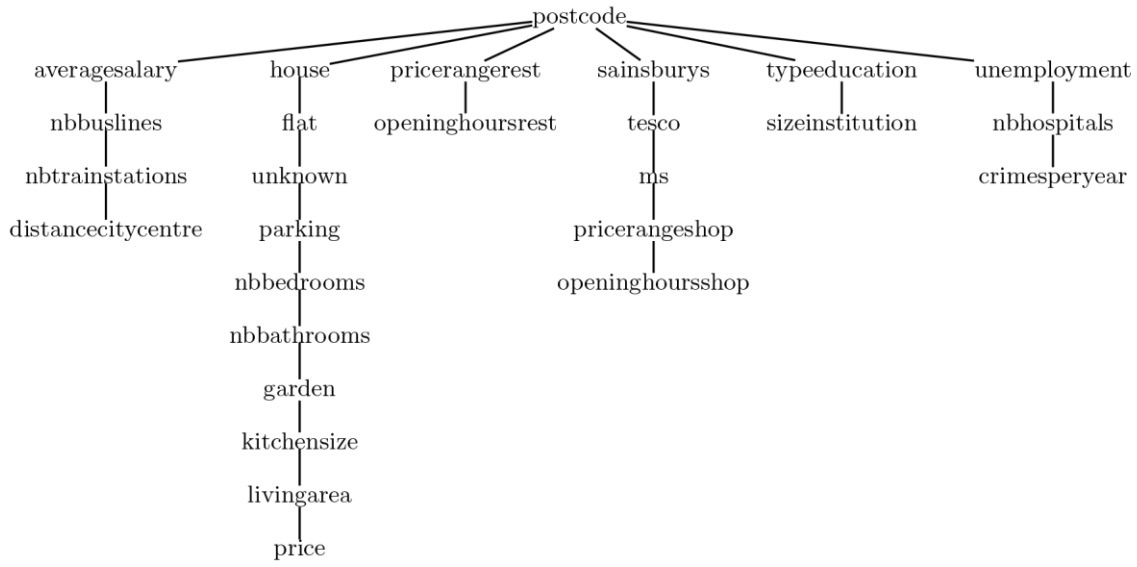


Figure 6.1: Optimal f-tree for Housing

This f-tree has a parameter $s(Q) = 2$ 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 $s(Q) = 2$ 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.

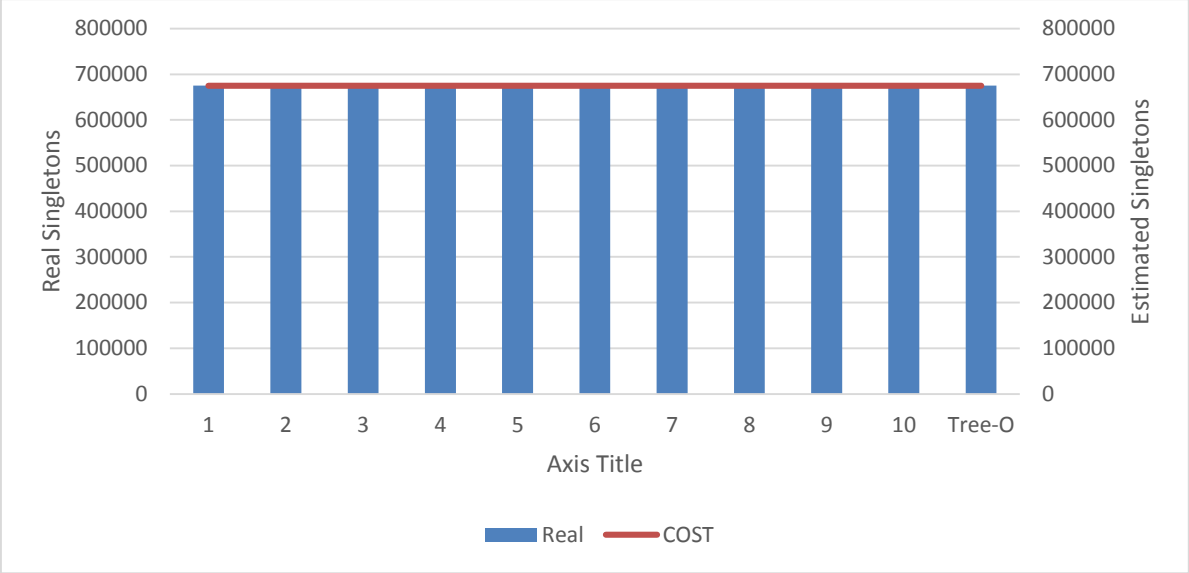


Figure 6.2: Real vs COST (Housing - 1)

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.

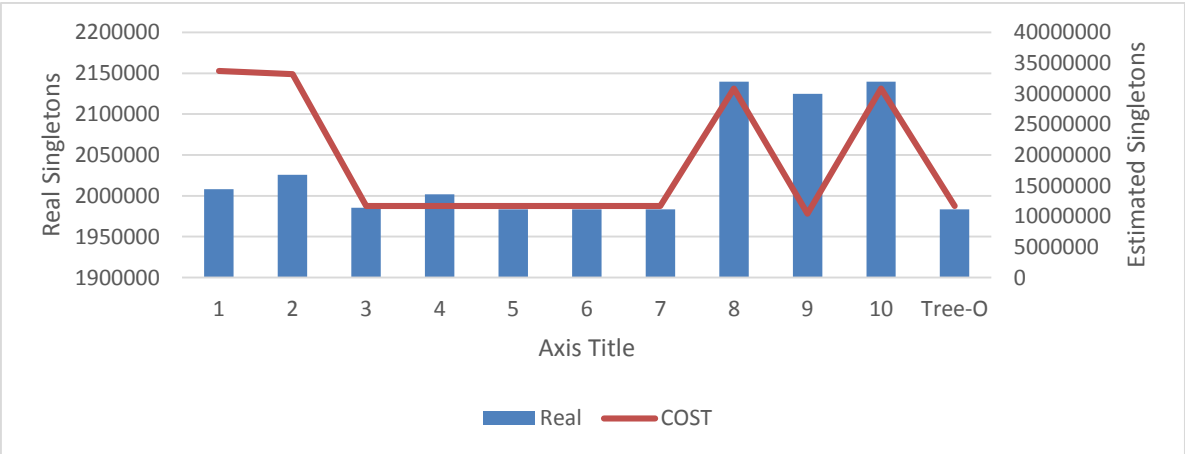


Figure 6.3: Real vs COST (Housing - 5)

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*.

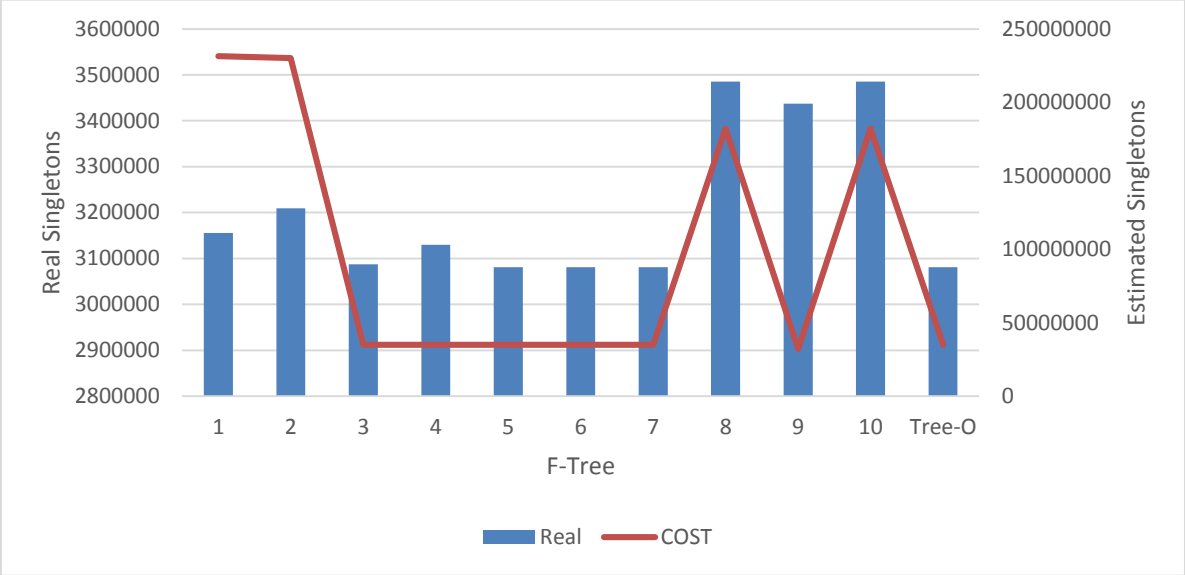


Figure 6.4: Real vs COST (Housing - 9)

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.

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.

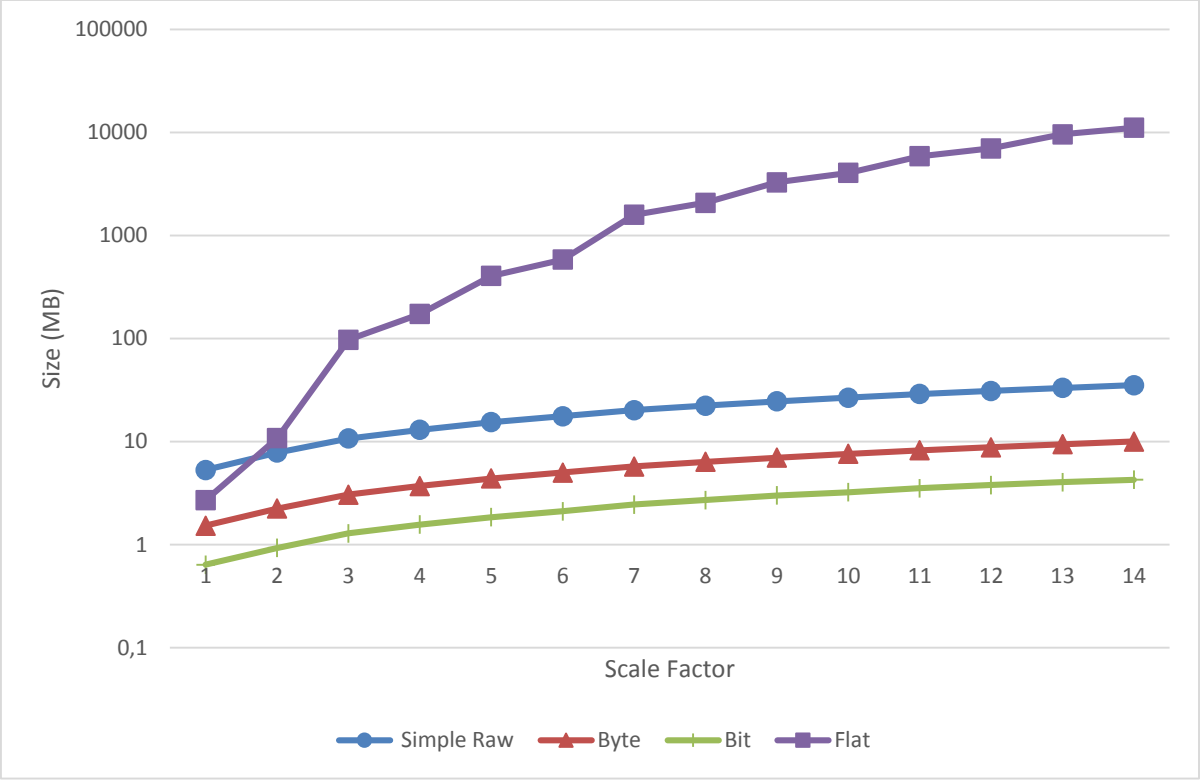


Figure 6.5: Serialization sizes against Flat serialization (Housing)

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.

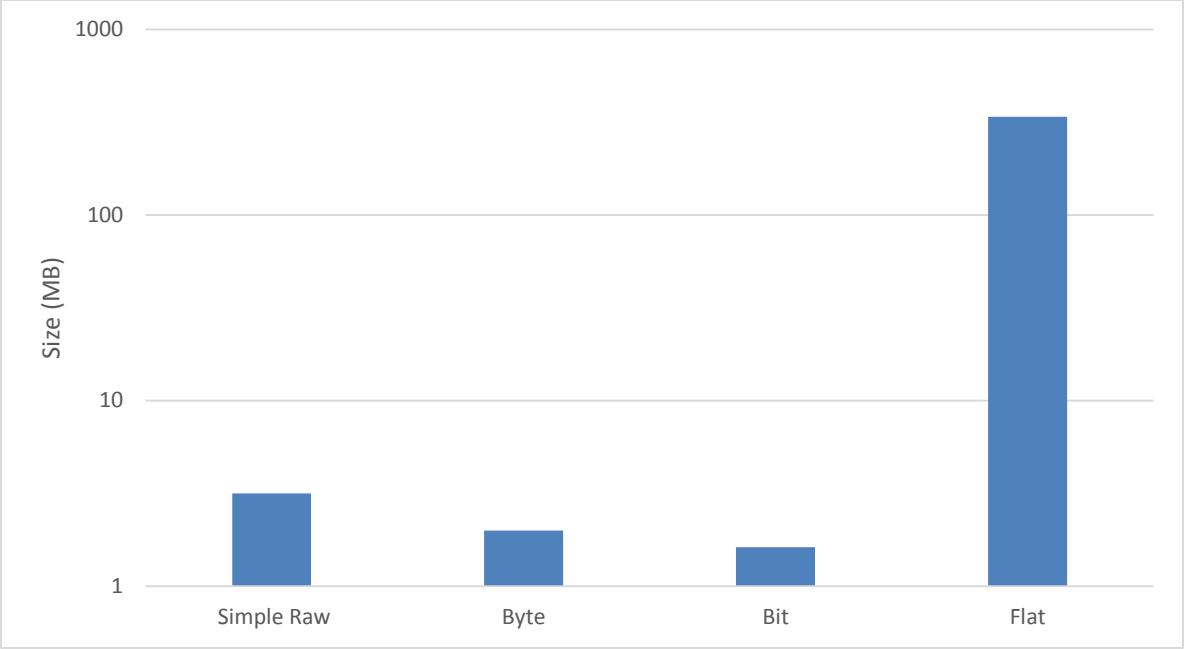


Figure 6.6: Serialization sizes against Flat serialization (US retailer)

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.

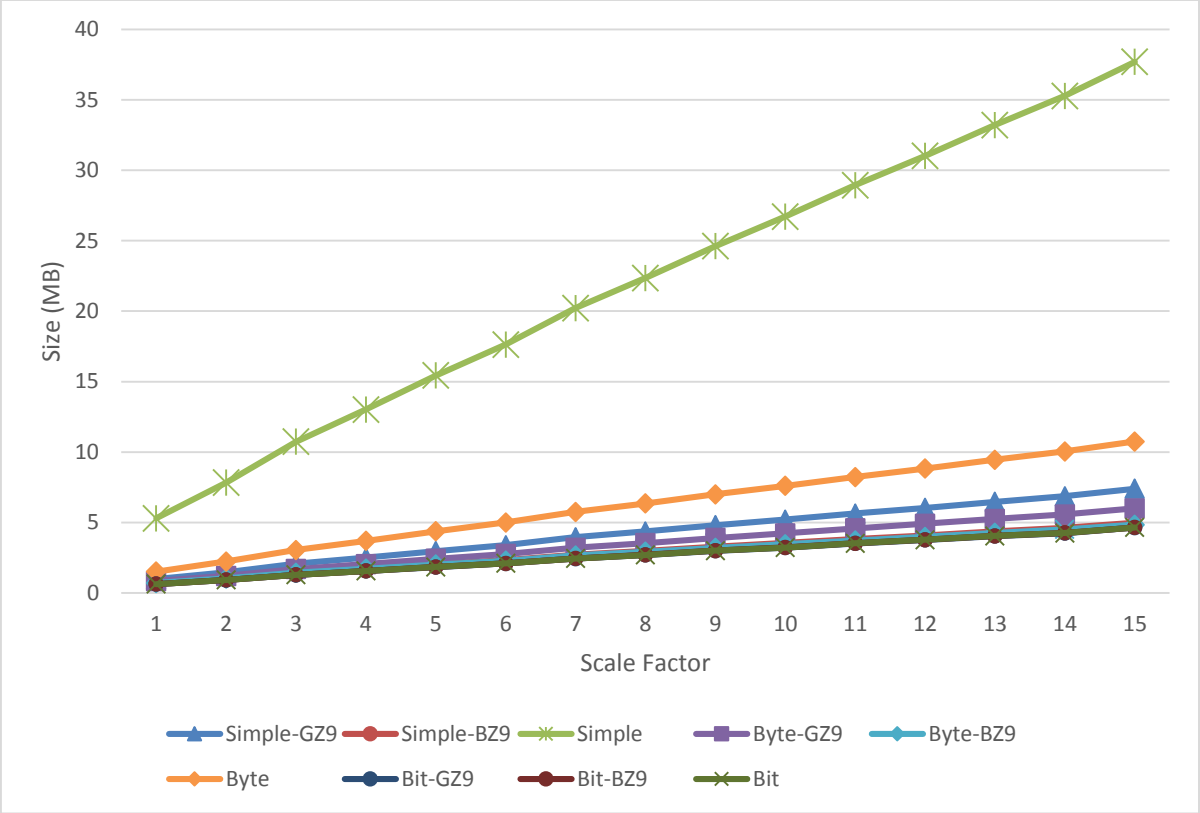


Figure 6.7: Compression GZIP and BZIP2 applied on our serializers

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.

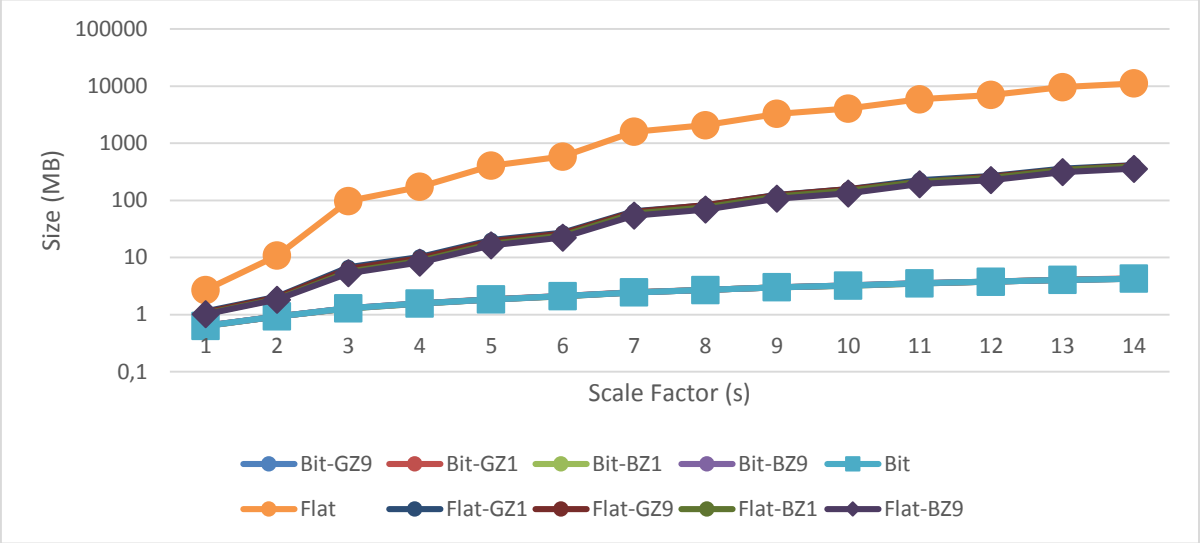


Figure 6.8: Compression GZIP and BZIP2 on Bit and Flat serializations (Housing)

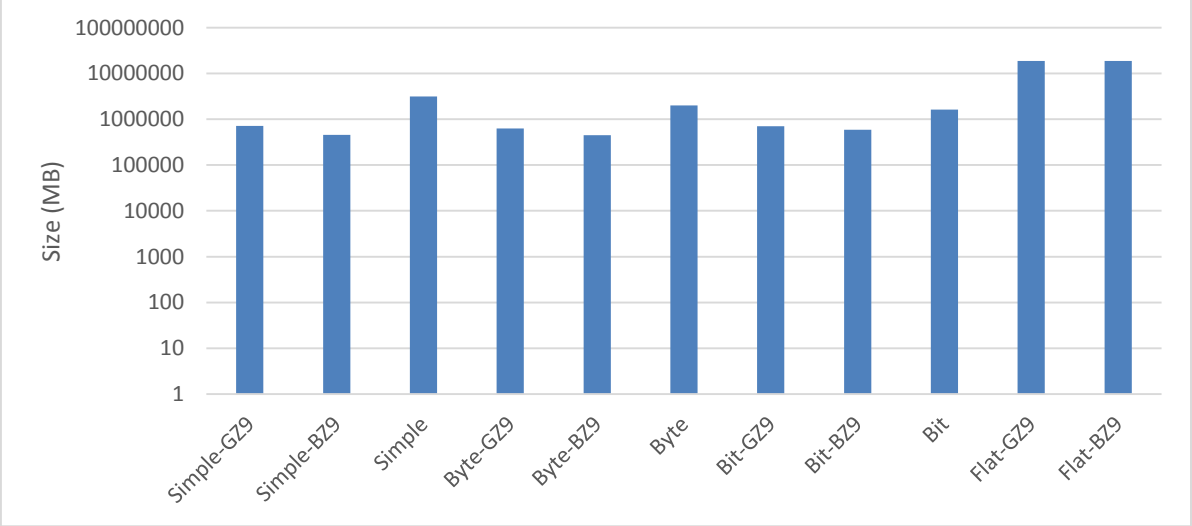


Figure 6.9: Compression GZIP and BZIP2 on Bit and Flat serializations (US retailer)

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.

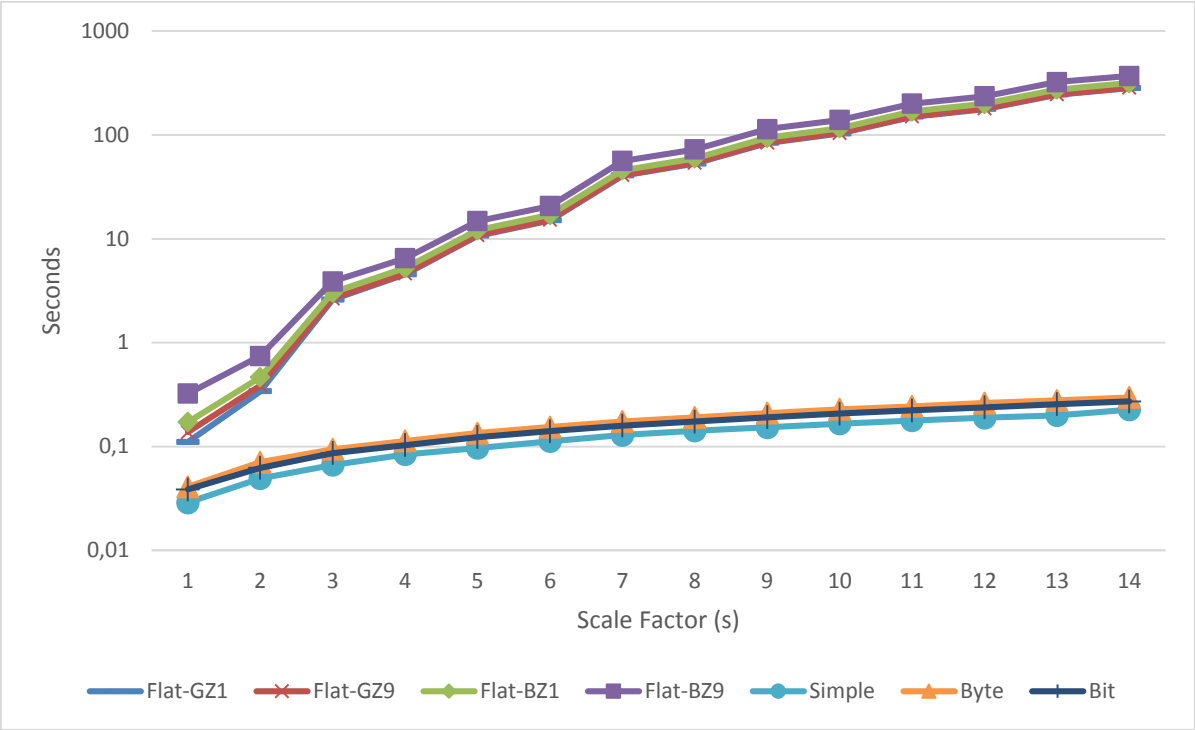


Figure 6.10: Serialization times with compression only on Flat (Housing)

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.

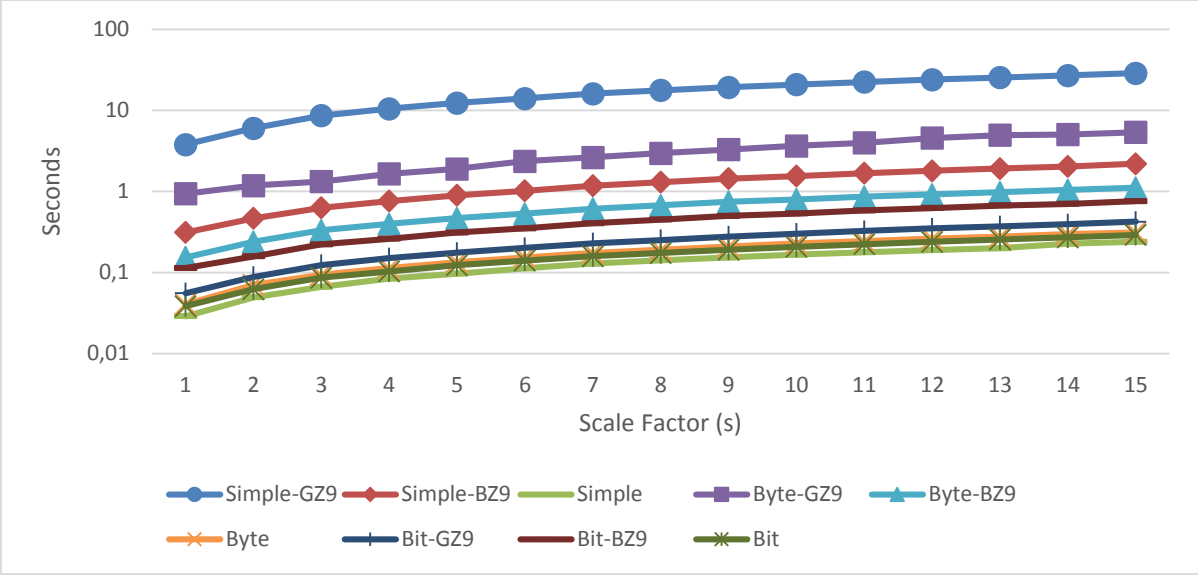


Figure 6.11: Serialization times with compression (Housing)

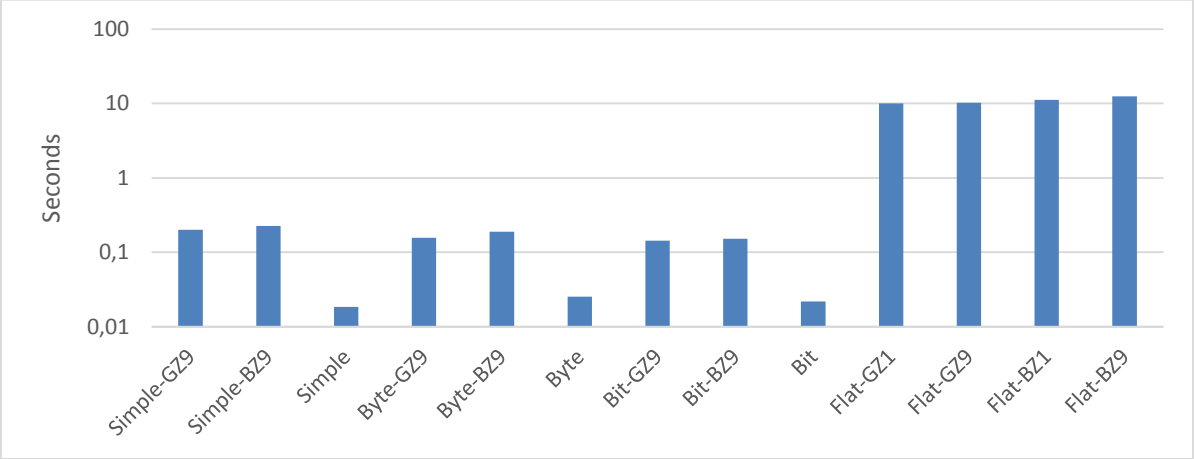


Figure 6.12: Serialization times with compression (US retailer)

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.

6.3.4 Deserialization times

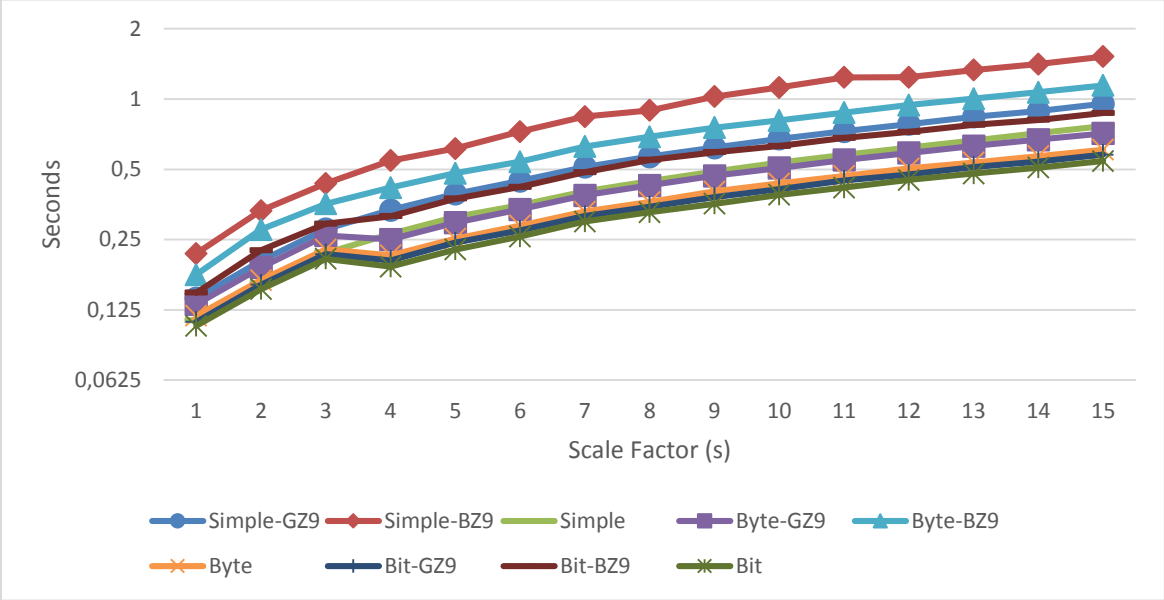


Figure 6.13: Deserialization times for our de-serializers (Housing)

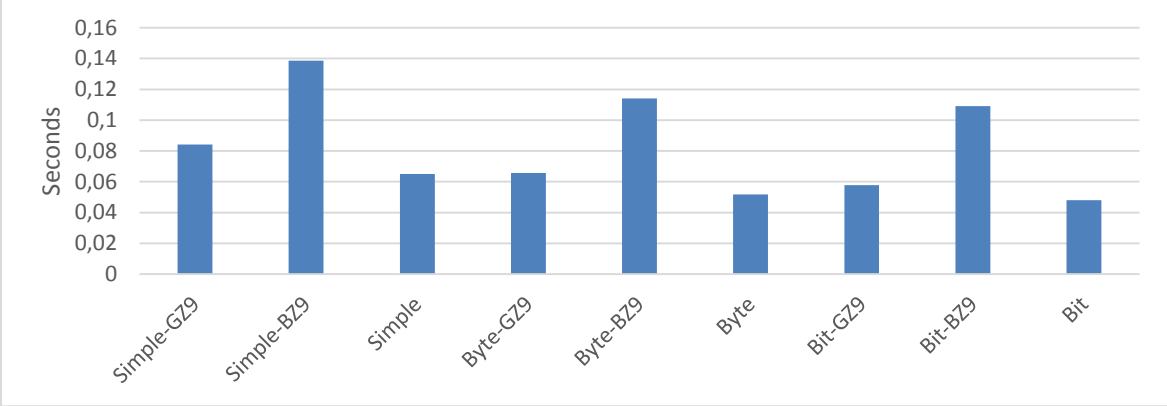


Figure 6.14: Deserialization times for our de-serializers (US retailer)

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.

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

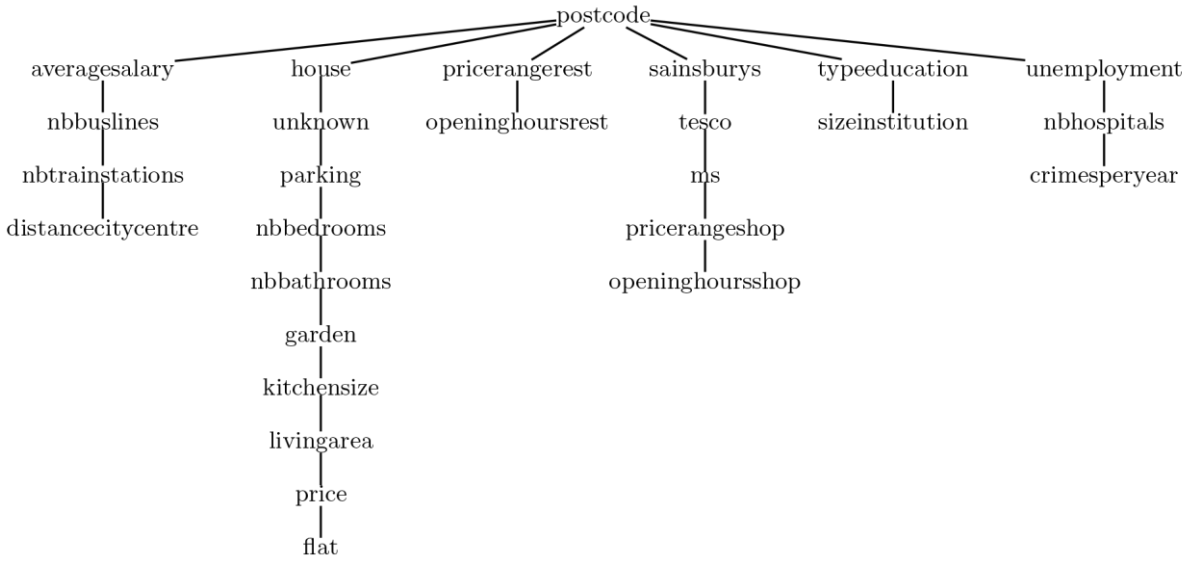
Conclusions and Future Work

Mini TOC

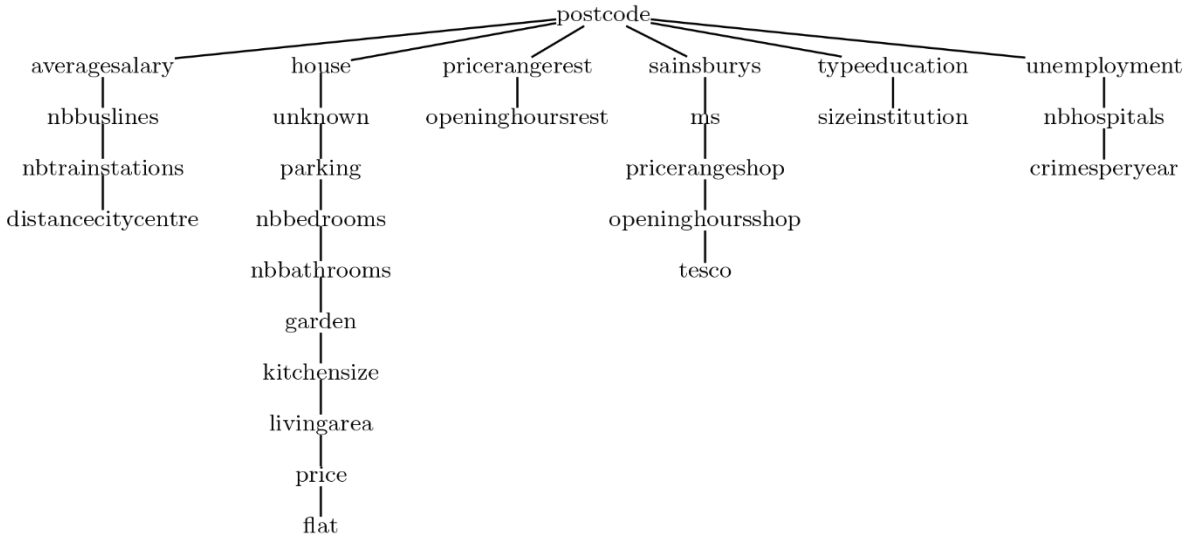
References

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.

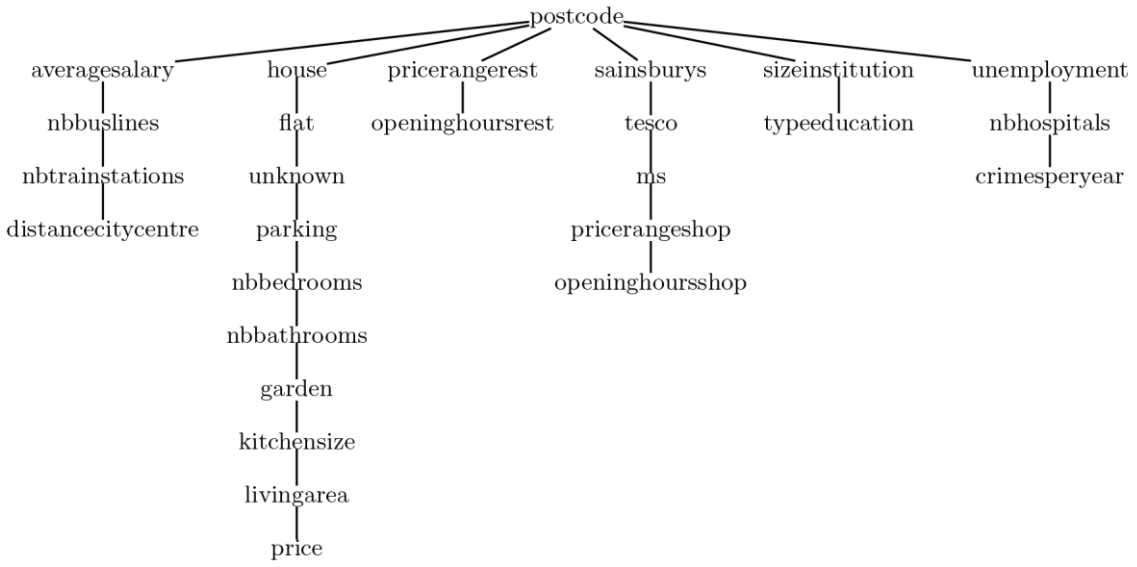
Appendix A



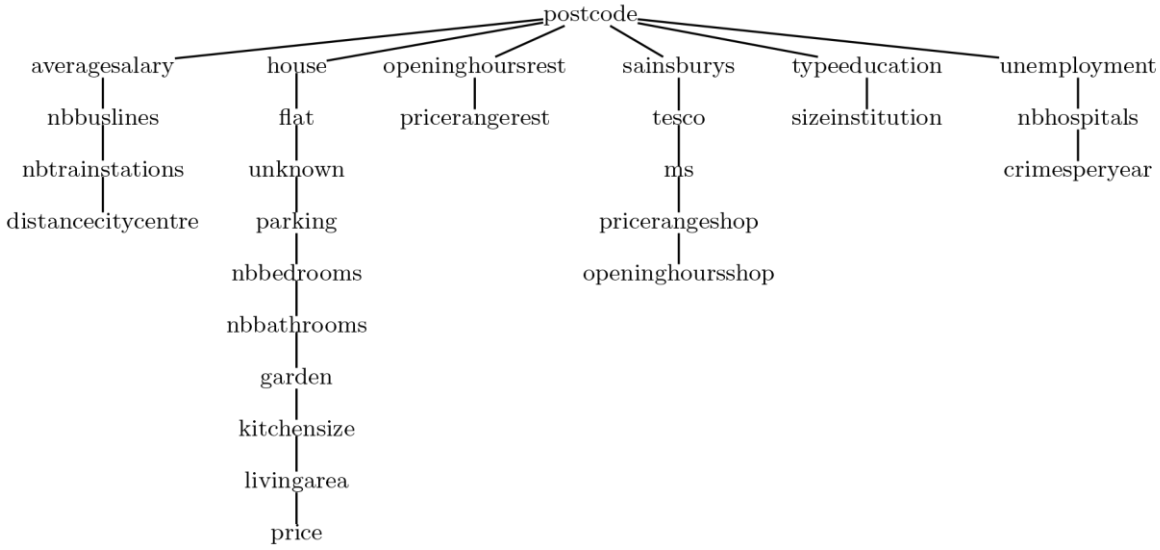
Appendix 1: F-Tree 1



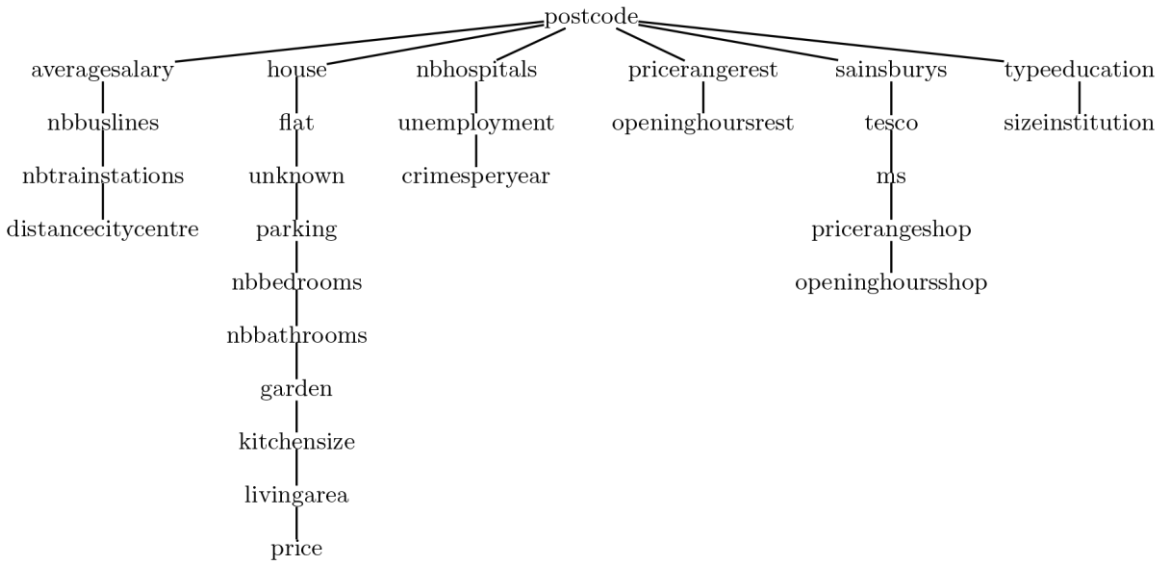
Appendix 2: F-Tree 2



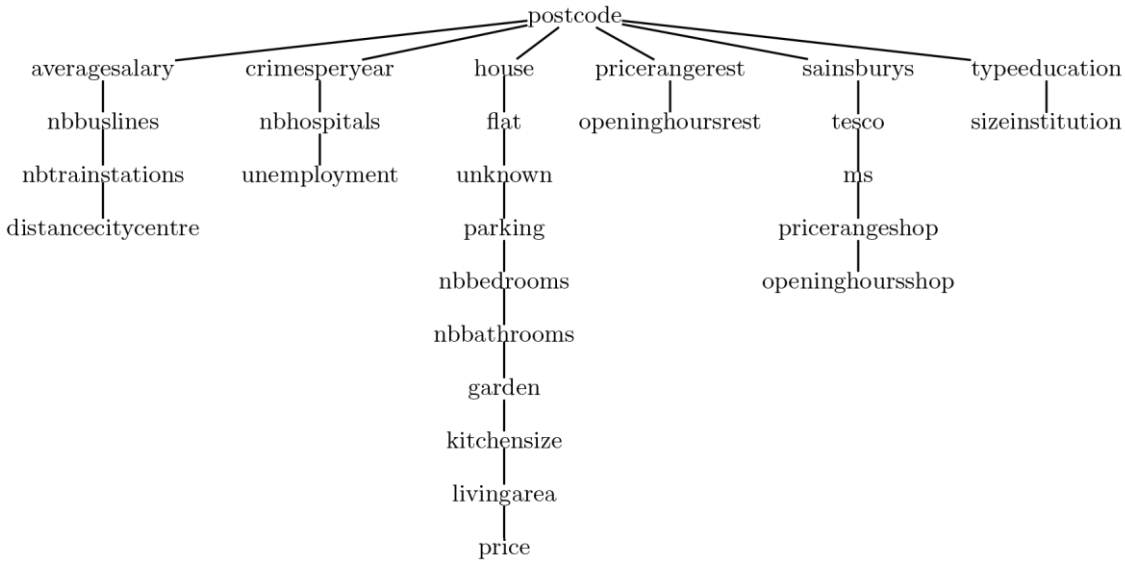
Appendix 3 : F-Tree 3



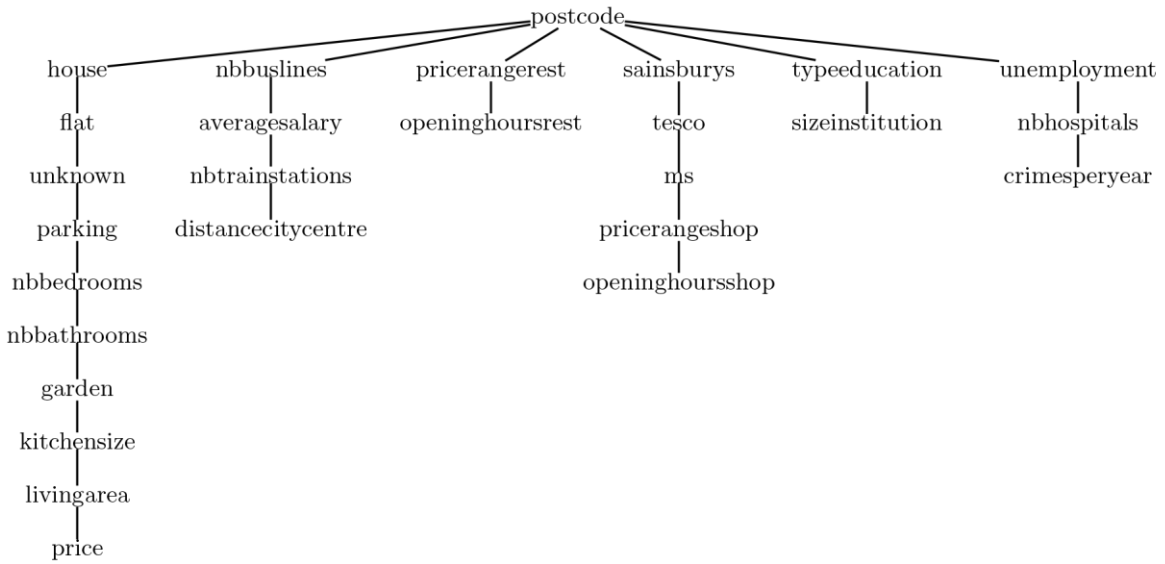
Appendix 4: F-Tree 4



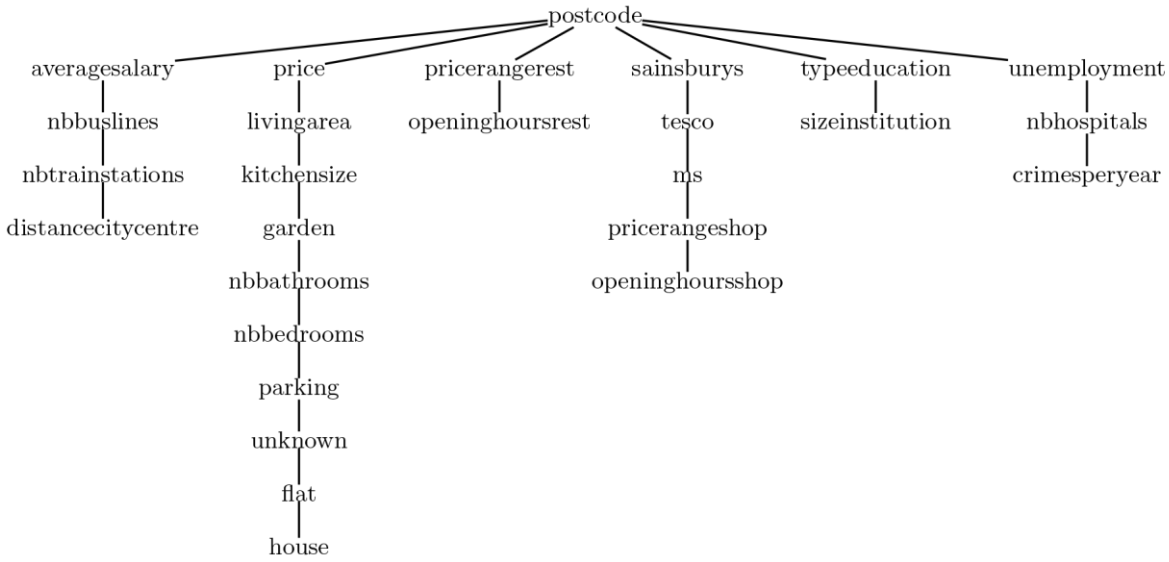
Appendix 5: F-Tree 5



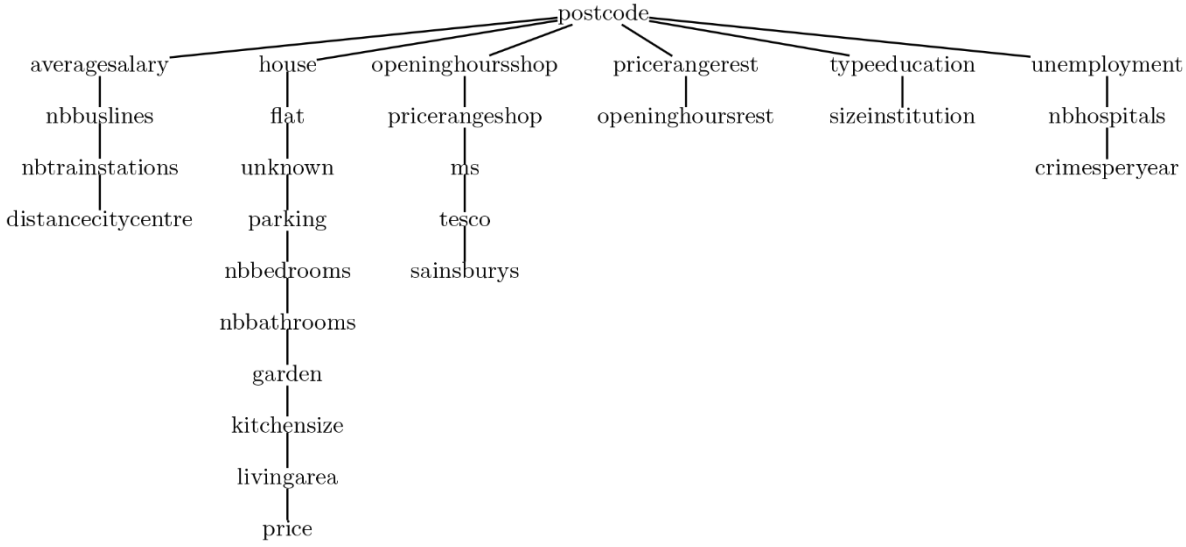
Appendix 6: F-Tree 6



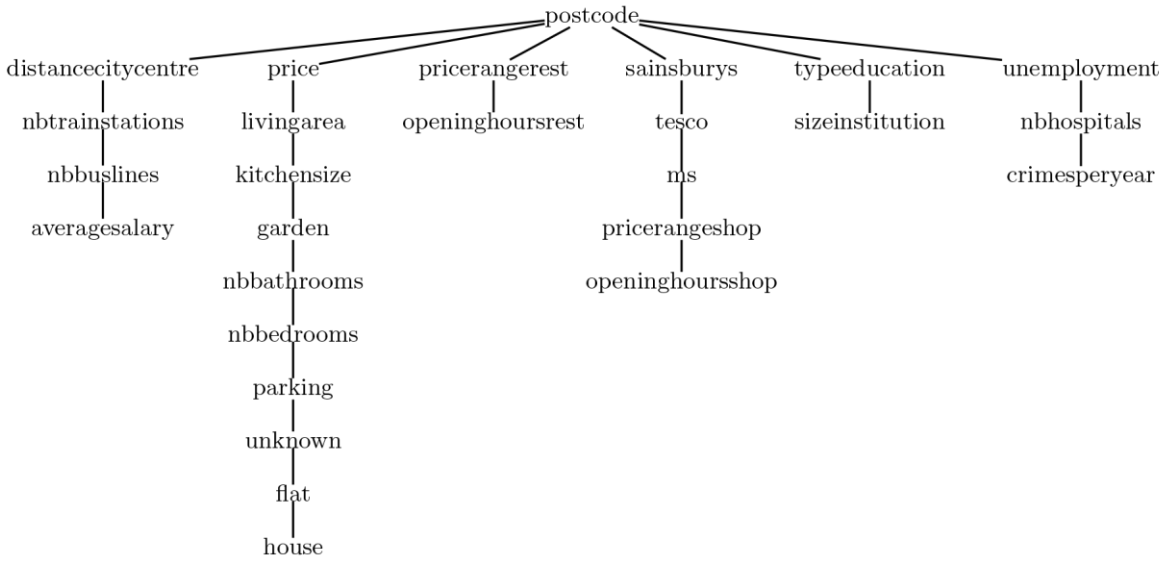
Appendix 7: F-Tree 7



Appendix 8: F-Tree 8



Appendix 9: F-Tree 9



Appendix 10: F-Tree 10

Algorithm X.Y: Algorithm title	
	// code here