Expectations

In all topologies, weight is an attribute of the edges and uniformly distributed from 1 to 9. Consequently, the selection:

```
weight < 2
```

will therefore remove around 89% of all edges. A lesser amount of edges immensely shortens the duration of any subsequent join process.
Since joining the entire vertex and edge set is a very costly operation, it is possible that queries with multiple, but restricted joins are faster than such, which contain fewer joins, but do not include any restrictions on the edge set.

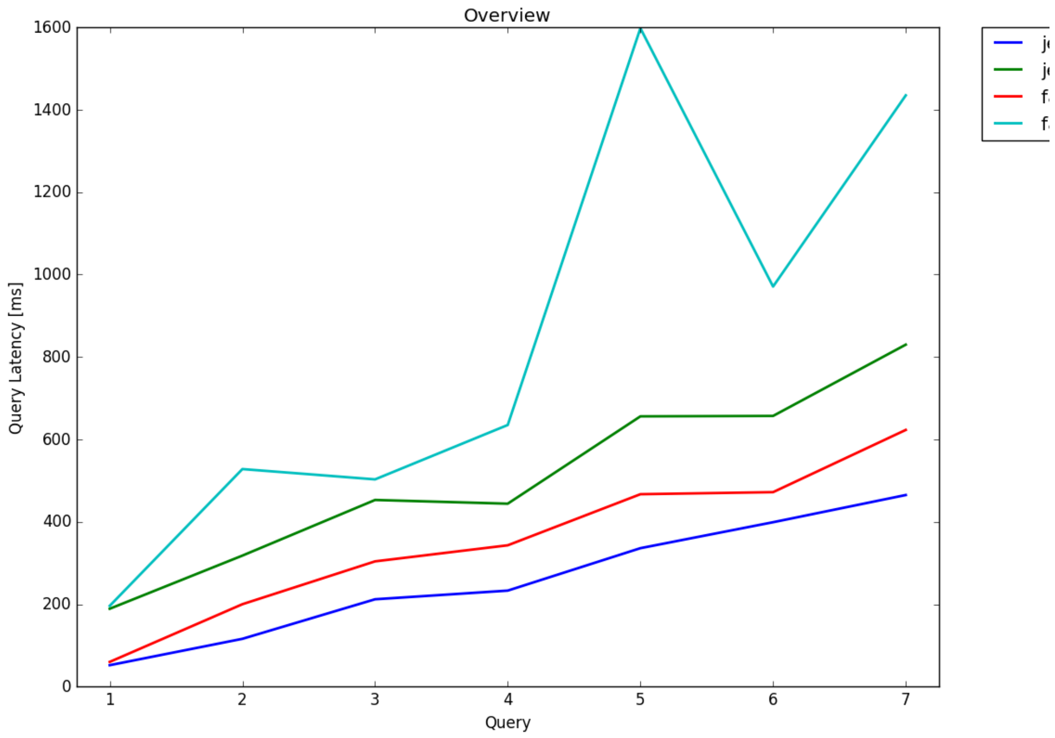
Another thing to consider is the fact that the attributes of vertices and edges are stored in a Vector of (String, Literal) tuples. We chose this suboptimal implementation since Timely Dataflow does not aprreciate HashMaps at all. However, in order to do a selection, this Vector has to be transformed back into a HashMap, which takes some time. Therefore the more selection a query contains, the slower it should be.

It should also be noted that one single edge in the query triggers two joins in the evaluation.Reason being that we have to join the edge collection twice with the vertex collection, in order to determine all adjacent vertices of a given vertex.

We ran seven different queries, with each subsequent query being more complex than the preceding one. For each query the resulting dataflow is drawn with the numbers on the edges indicating the size of (intermediate) result set.

Throughout all seven queries, the following pattern is noticeable:
The influence of the number of workers behaves unexpectedly, especially for queries 1,4 and 6. In a perfect world, the query latency would be inversely proportionalto the number of workers. However, we only observe this behaviour for query 5, and only for the fattree topologies.
More often than not, an increase of worker threads leads to a higher latency, especially when the latency is already quite low, in the 50 to 200 millisecond range. We reason this happens because at that level, communication. This

The following pages contain detailed boxplots and the dataflows for each of the seven queries. **Overview**


Overview
Query Latency [ms]
Query
1600
1400
1200
1000
800
600
400
200
0
1
2
3
4
5
6
7
j
j
f
f

Query 1

```
SELECT * WHERE u.label() = 'switch', u.position = 'access'
```

Peak Memory: 377'840 kb
Result size: 1'152
2 simple selections, 0 joins. One pass through the vertices collection is enough to produce the result.

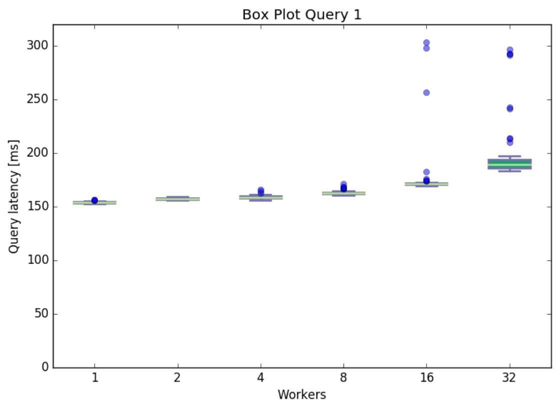


Figure 1: Large Fattree

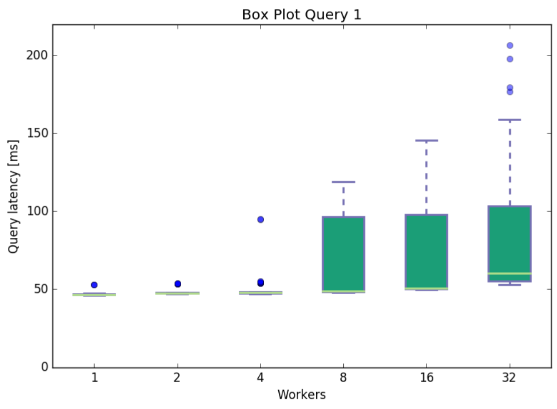


Figure 2: Small Fattree

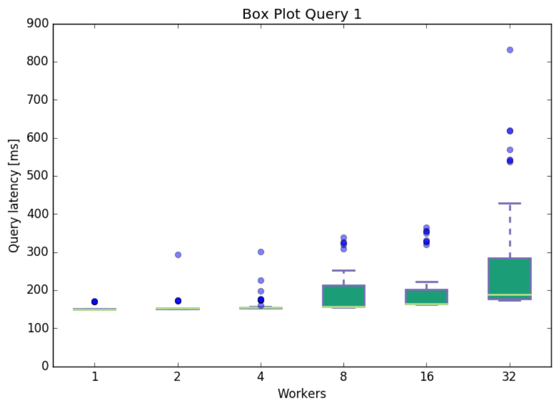


Figure 3: Large Jellyfish

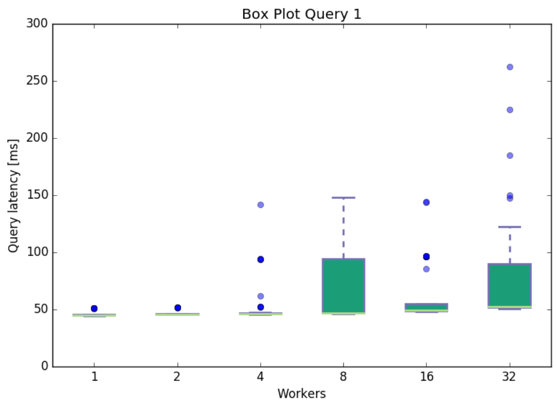
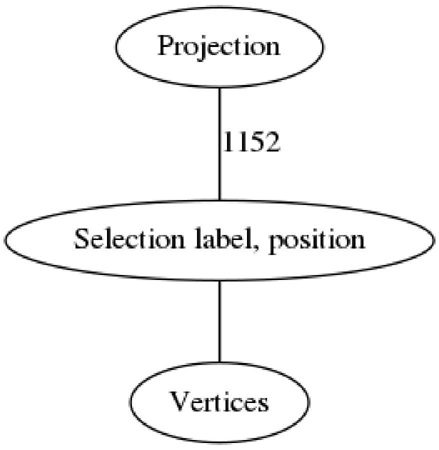


Figure 4: Small Jellyfish


Projection
1152
Selection label, position
Vertices

Query2

```
SELECT * WHERE (n) -[e with weight < 4]-> (m)
```

Peak Memory: 3’597’452 kb
Result size: 36’542
2 Joins and 1 Selection. The joins are not very expensive since we only use about 30% of the edges.

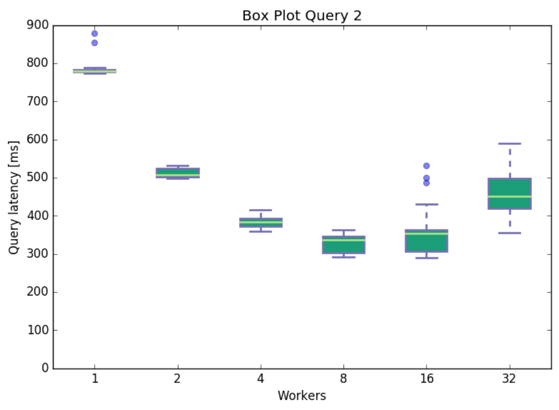


Figure 5: Large Fattree

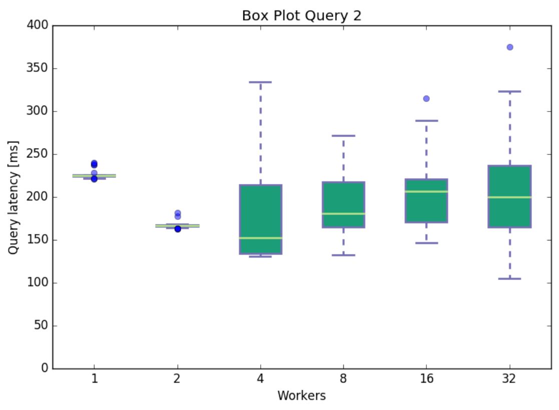


Figure 6: Small Fattree

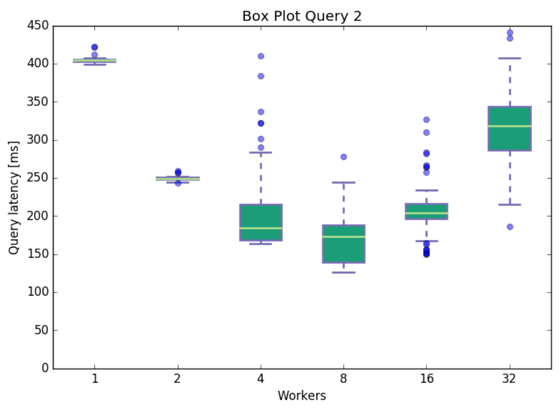


Figure 7: Large Jellyfish

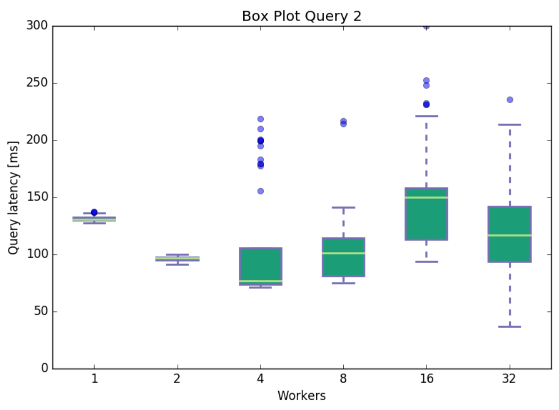
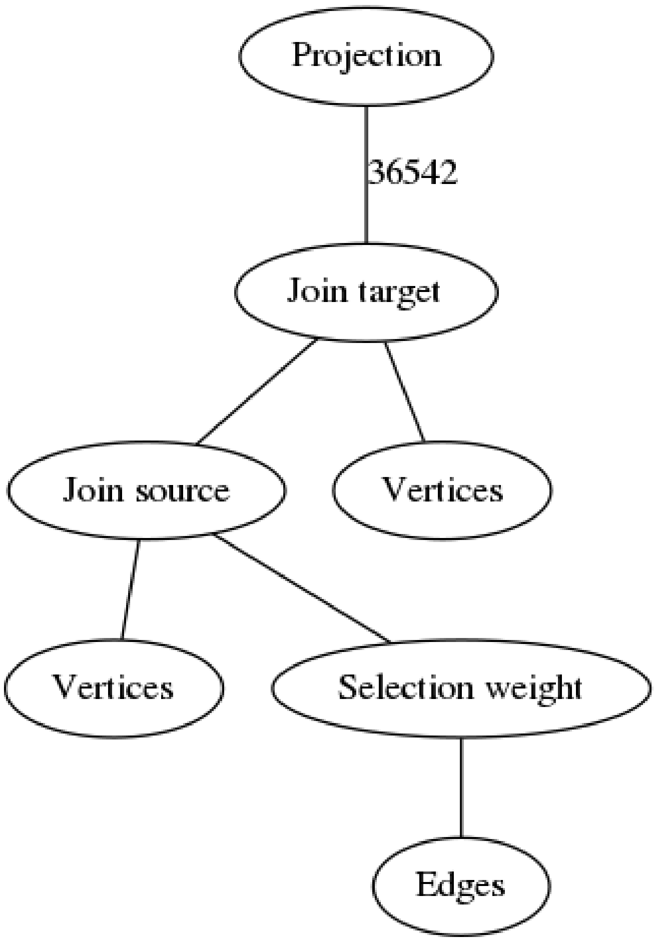


Figure 8: Small Jellyfish


Projection
36542
Join target
Join source
Vertices
Vertices
Selection weight
Edges

Query3

```
SELECT * WHERE (n:switch) -> (m with position = 'distribution')
```

Peak Memory: 1'433'376 kb
Result size: 55'296
2 Joins and 2 Selections. I expected this query to be a little bit slower since we are using all the edges in the joins, but it is only a tiny bit slower than Query 2.

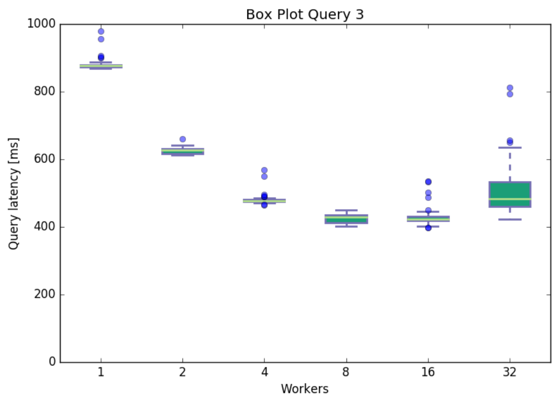


Figure 9: Large Fattree

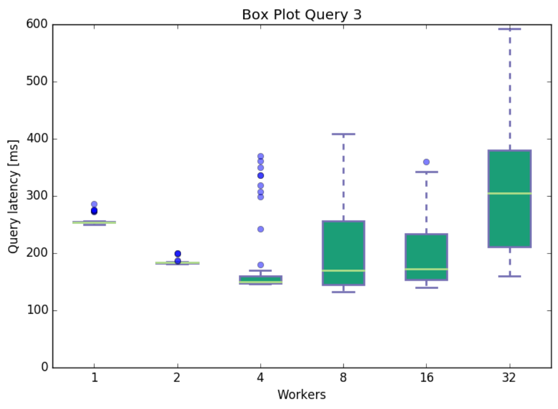


Figure 10: Small Fattree

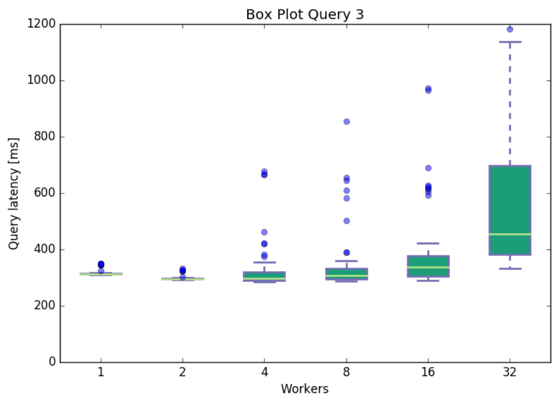


Figure 11: Large Jellyfish

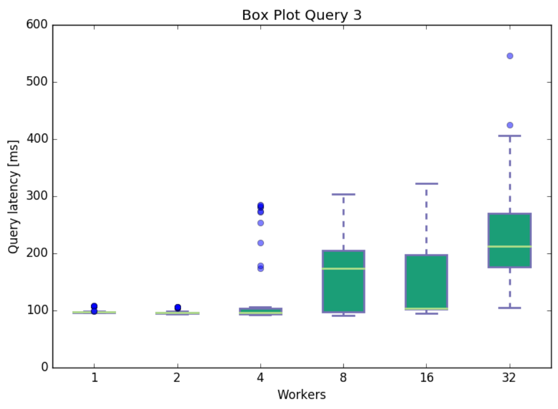
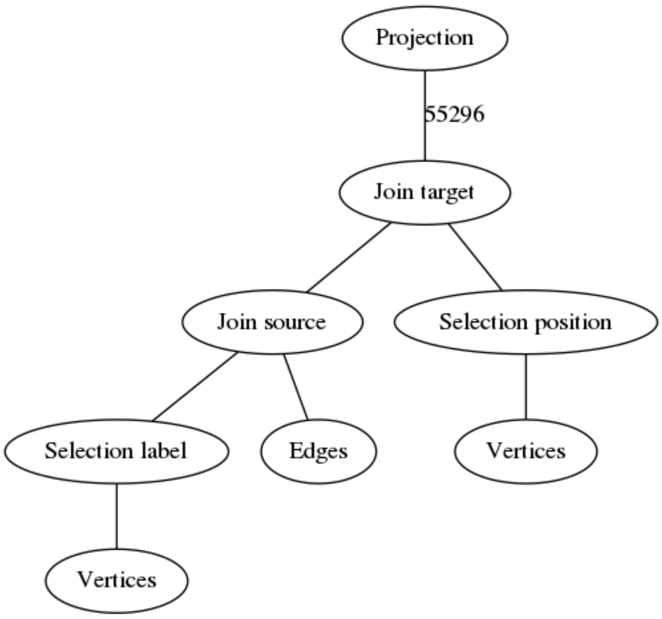


Figure 12: Small Jellyfish


Projection
55296
Join target
Join source
Selection position
Selection label
Edges
Vertices
Vertices

Query4

```
SELECT * WHERE (n with position = 'distribution')
-[e with weight > 8]-> (m with position = 'access')
```

Peak Memory: 1'114'896 kb
Result size: 3'097
2 Joins and 3 Selections. Even though this query has more constraints than Query 2 and 3, it is faster since we only use 10% of the edges in the joins.

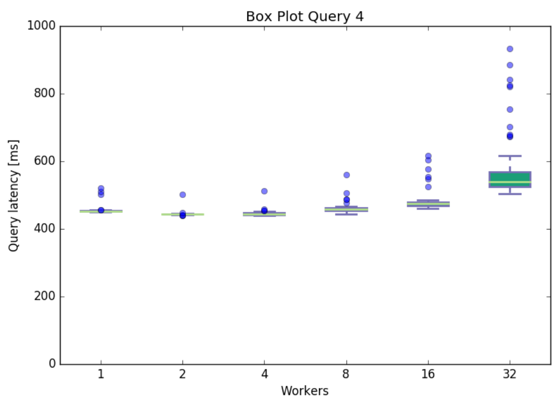


Figure 13: Large Fattree

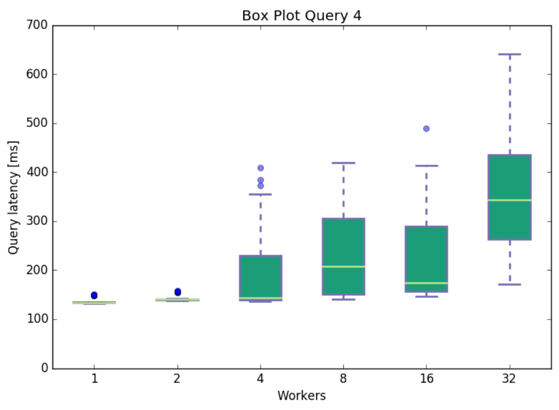


Figure 14: Small Fattree

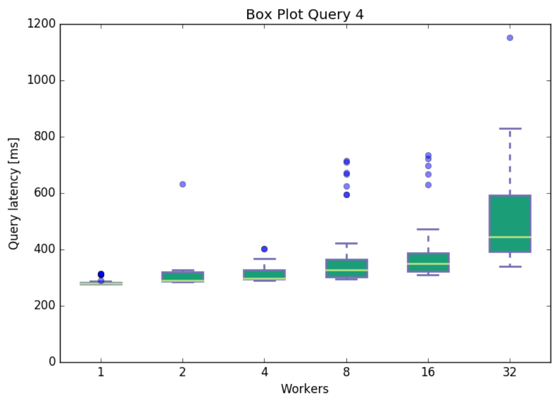


Figure 15: Large Jellyfish

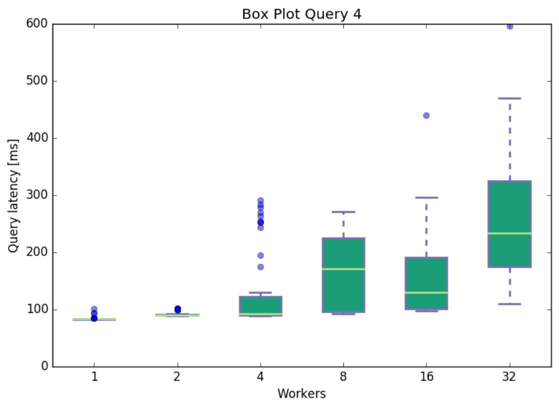
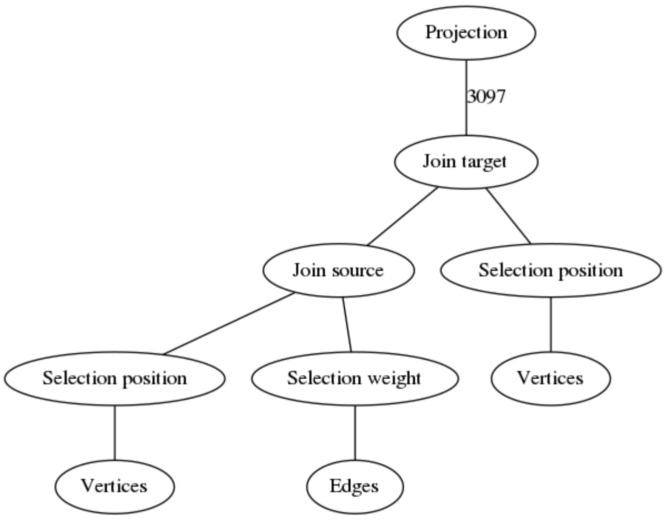


Figure 16: Small Jellyfish


Projection
3097
Join target
Join source
Selection position
Selection position
Selection weight
Vertices
Vertices
Edges

Query5

```
SELECT * WHERE (u WITH position = 'access') -> (v WITH position = 'distribution')
 -> (w WITH position = 'core')
```

Peak Memory: 3'797'976 kb
Result size: 663'552
4 Joins and 3 Selections. The last 3 queries all contain multiple query edges and are much slower. Since there is no selection on the edge, this particular query is quite slow even though it has "only" 4 joins.

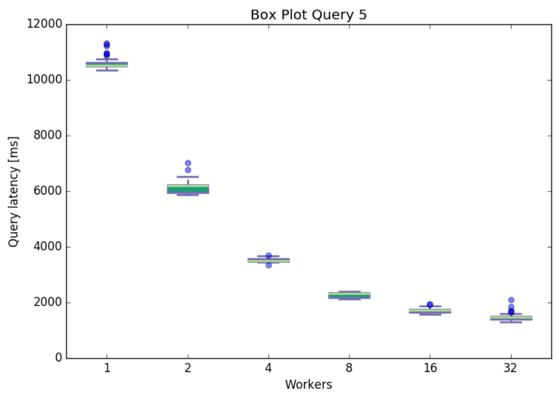


Figure 17: Large Fattree

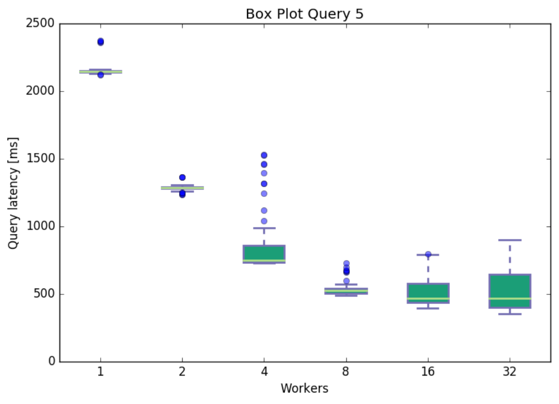


Figure 18: Small Fattree

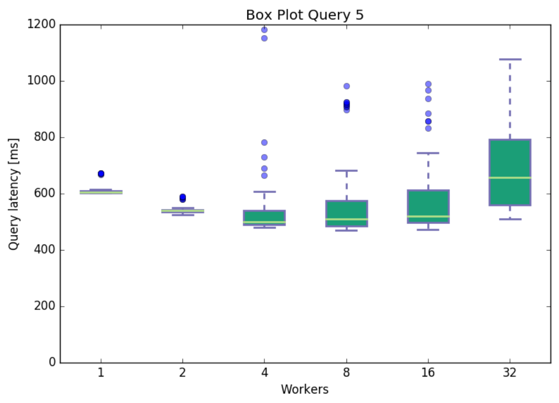


Figure 19: Large Jellyfish

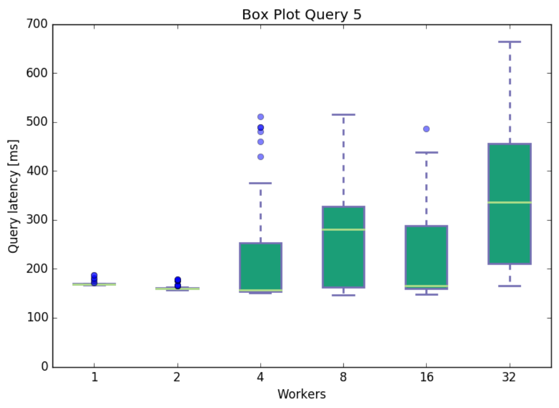
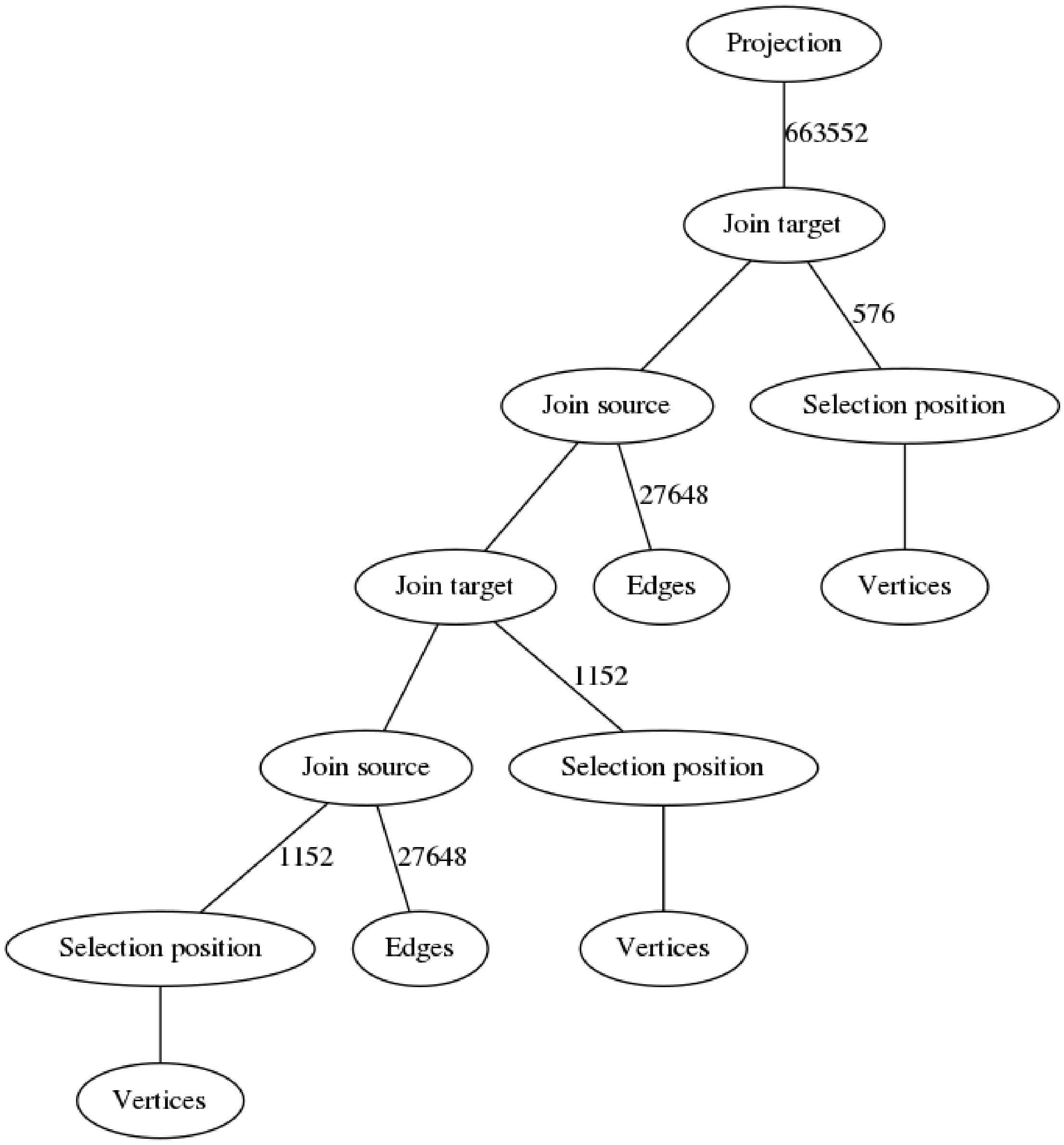


Figure 20: Small Jellyfish


Projection
663552
Join target
576
Join source
Selection position
27648
Join target
Edges
Vertices
1152
Join source
Selection position
1152
27648
Selection position
Edges
Vertices
Vertices

Query6

```
SELECT * WHERE (u WITH position = 'access')
 -[e with weight  < 2]-> (v WITH position = 'distribution')
 -[f with weight > 8]-> (w WITH position = 'core')
```

Peak Memory: 1'573'940 kb
Result size: 8'334
4 Joins and 5 Selections. Like query 4, the evaluation time goes down as we add more constraints on the edges. Since we have a lot less tuples in the 3rd and 4th join, this query is faster than previous one.

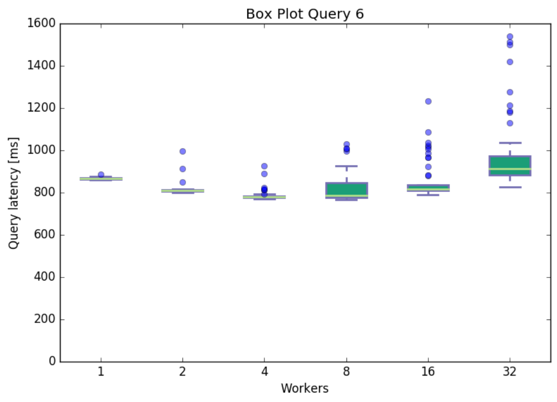


Figure 21: Large Fattree

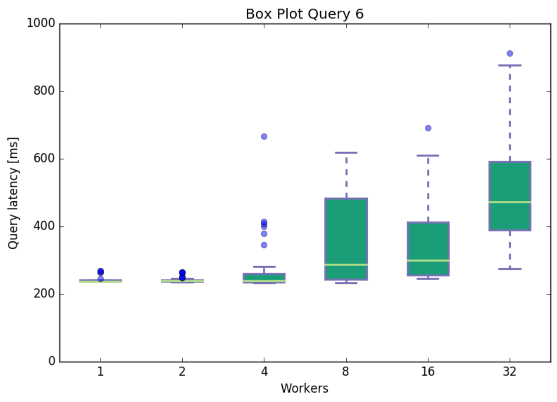


Figure 22: Small Fattree

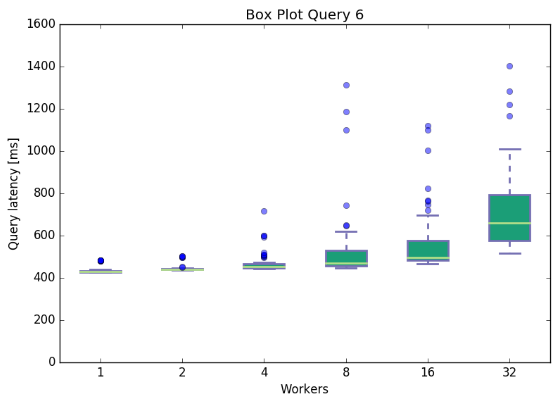


Figure 23: Large Jellyfish

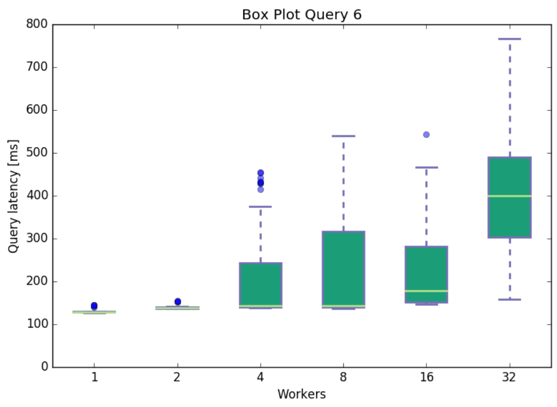
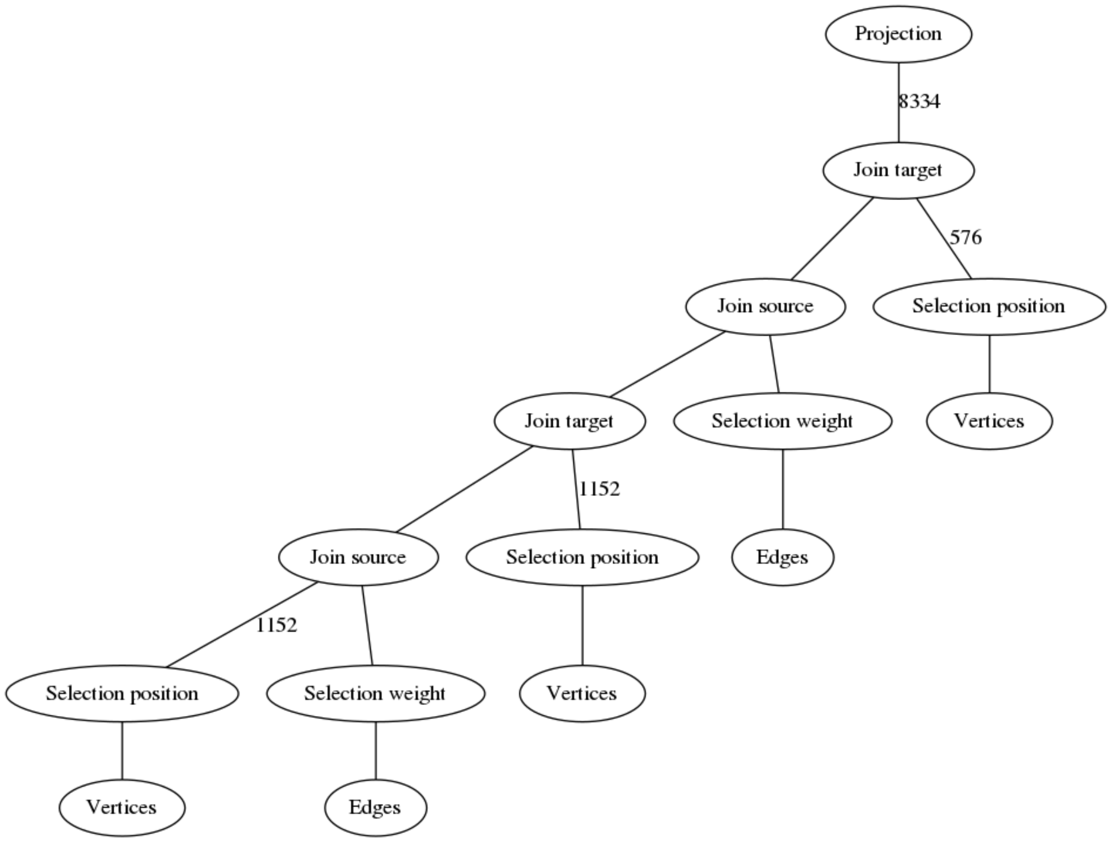


Figure 24: Small Jellyfish


Projection
8334
Join target
576
Join source
Selection position
Join target
Selection weight
Vertices
1152
Join source
Selection position
Edges
1152
Selection position
Selection weight
Vertices
Vertices
Edges

Query7

```
SELECT * WHERE (u WITH position = 'access')
 -[e with weight  < 2]-> (v WITH position = 'distribution')
 -[f with weight  < 2]-> (w WITH position = 'core')
 -[g with weight  < 2]-> (x WITH position = 'distribution')
```

Peak Memory: 2'210'824 kb
Result size: 49'956
6 Joins and 7 Selections. This is probably the most interesting result. We add two more joins to the query, but we restrict the number of edges used in all the joins dramatically. This consequently leads to a surprisingly low evaluation time, almost on the level as query 5. This demonstrates how big the impact of the joins is for the evaluation time.

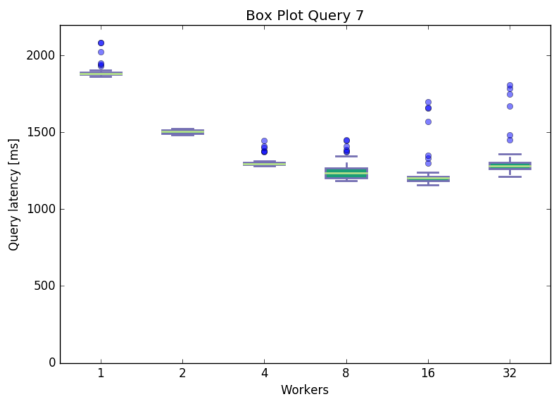


Figure 25: Large Fattree

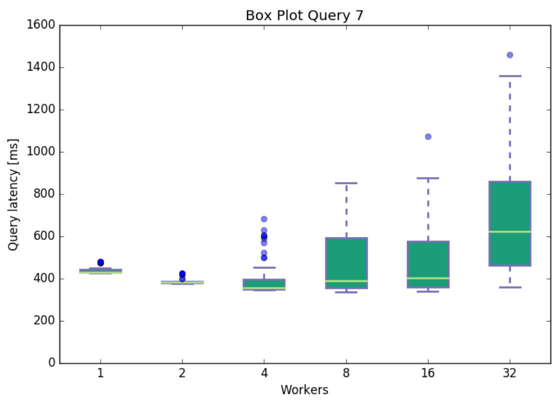


Figure 26: Small Fattree

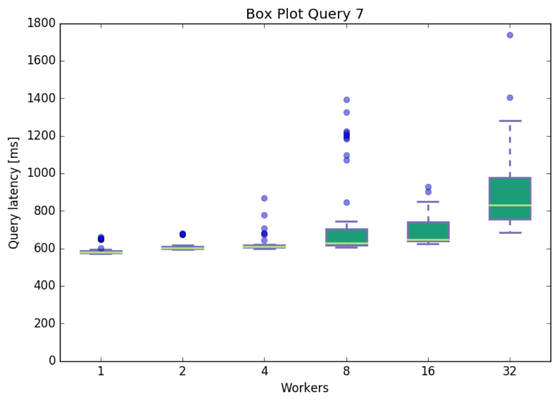


Figure 27: Large Jellyfish

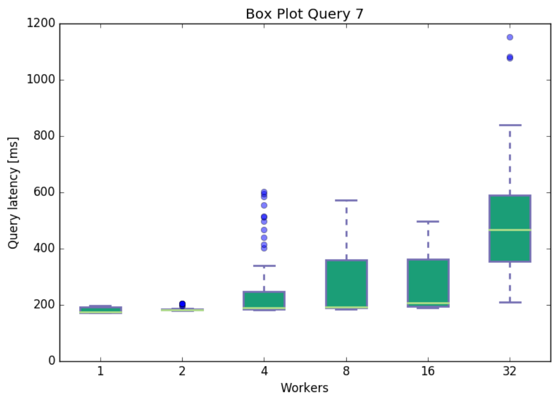
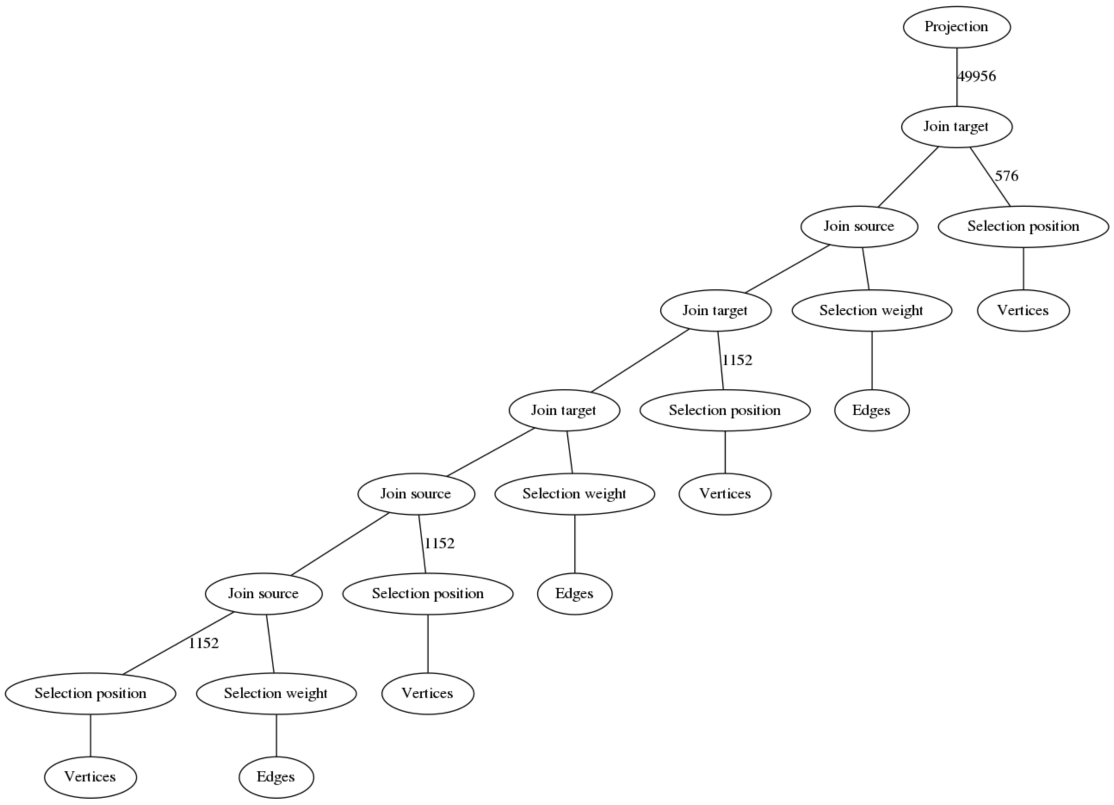


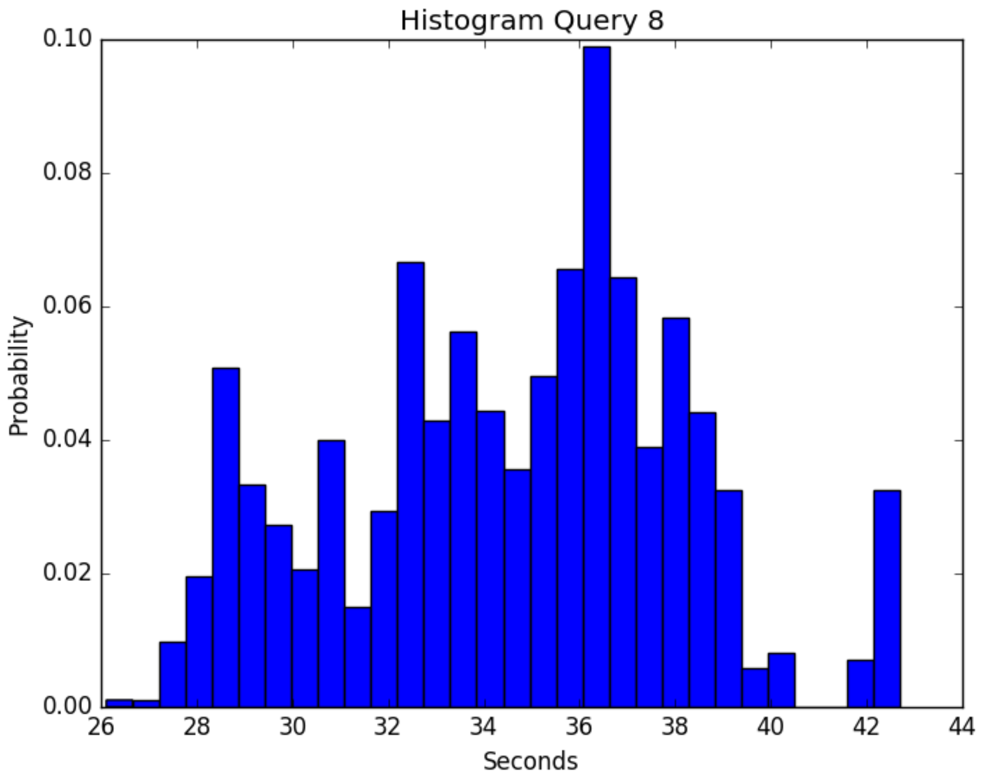
Figure 28: Small Jellyfish



Projection
49956
Join target
576
Join source
Selection position
Join target
Selection weight
Vertices
1152
Join target
Selection position
Edges
Join source
Selection weight
Vertices
1152
Join source
Selection position
Edges
1152
Selection position
Selection weight
Vertices
Vertices
Edges

Query 8

```
SELECT * WHERE (u WITH position = 'access') -> (v WITH position = 'distribution')
 -> (w WITH position = 'core') -> (x WITH position = 'distribution')
```



6 Joins and 4 Selections. I have not included this query in the figure because the evaluation times are around 30 to 40 seconds when run with 48 workers. When using only a single worker I have measured evaluation times over 10 minutes. This is due to the fact that we join 6 times with the full edge collection.