1 Hardware

All the experiments were conducted on an ETH server (exact model?). This machine possesses 4 x 12 AMD Opteron 6174 processors, each one running at 2.2 GHz. The operating System was Debian 7.0 and the installed memory was 128 GB of RAM.

2 Topologies

In the experiment, four different topologies were used:

Large Fattree The large fattree topology is the largest one of the four topologies. It contains 55'000 vertices, of which 2'880 are switches. Furthermore the graph posses a total of 50'000 bidirectional edges. Since one bidirectional edge is represented by two directed edges, there are a total of 100'000 edges in the collection.

Small Fattree The small fattree topology has the same vertices to edge ratio as the large fattree, but it is overall much smaller. It contains 17'665 vertices, of which 1'280 are switches. Furthermore the graph possesses a total of 16' 400 bidirectional edges. (needs better formulation)

Large Jellyfish The large jellyfish topology matches the large fattree in terms of vertices, but contains far less edges. It contains 56'160 vertices, of which 864 are switches. Furthermore the graph possesses a total of 6'910 bidirectional edges. (needs better formulation)

Small Jellyfish The small jellyfish topology is the smallest one of the four topologies. It contains 16'770 vertices, of which 390 are switches. Furthermore the graph possesses a total of 2'145 bidirectional edges. (needs better formulation)

3 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.

Furthermore, it is important to consider the fact that the attributes of vertices and edges in a Differential Dataflow Collection are stored in a Vector of (String, Literal) tuples. We chose this suboptimal implementation since Differential Dataflow does not aprreciate HashMaps at all. However, in order to do a selection on either set, this Vec-

tor has to be transformed back into a HashMap for each edge respectively vertex. This process can be quite timeconsuming, therefore the more selection a query contains, the slower it expected to 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.

In total, we ran seven different queries, with each subsequent query being more complex than the preceding one. For each query the resulting dataflow is drawn on the following pages. The numbers on the edges indicate the size of (intermediate) result set on the large fattree.

4 Overview

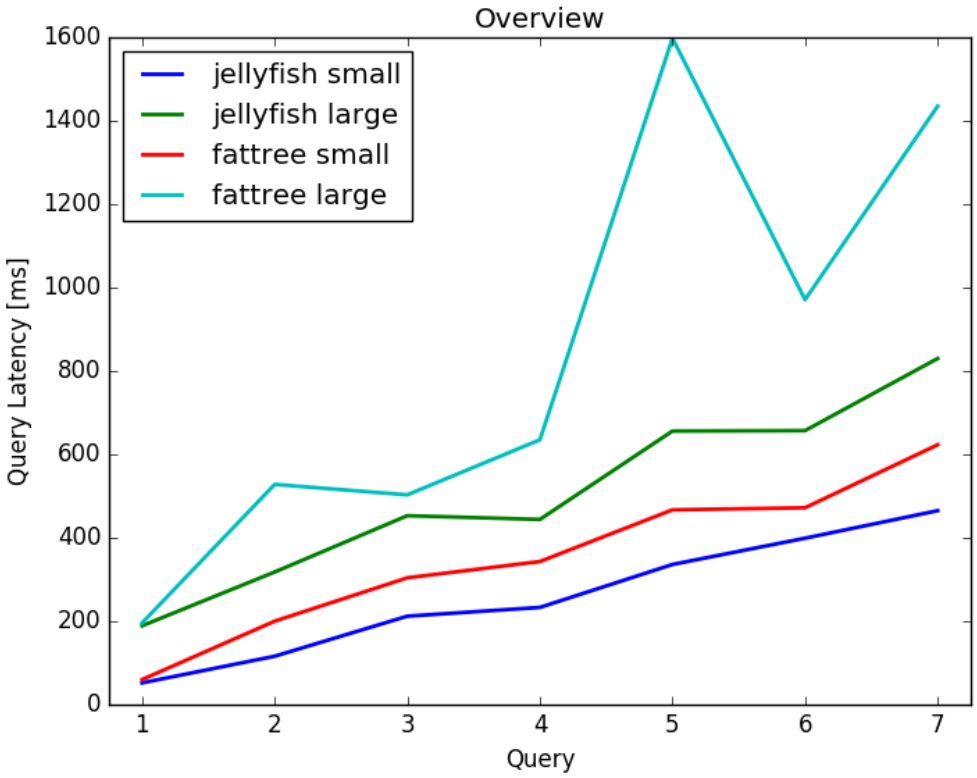

Figure 1: An overview of the seven queries, run with 32 workers

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, around the 50 to 200 millisecond range. We reason this happens because at that level, the communication overhead dominates the time savings achieved by parallelism. This is claim is additionally supported by the fact, that query 5 has one of the longest latencies, ranging up to 10 seconds. When evaluating queries with lower latency, the contention between the workers increases and we frequently observe that the best performance is achieved when running the query with just a single worker.
The following pages contain detailed boxplots and the dataflows for each of the seven queries.

5 Query 1

```
SELECT u.name 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.

Dataflow

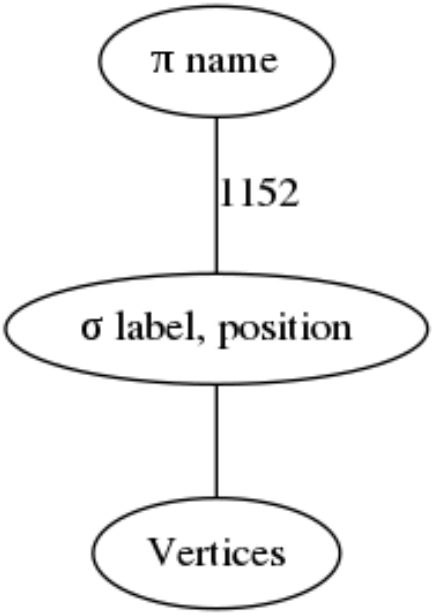

Figure 2: Dataflow Query1

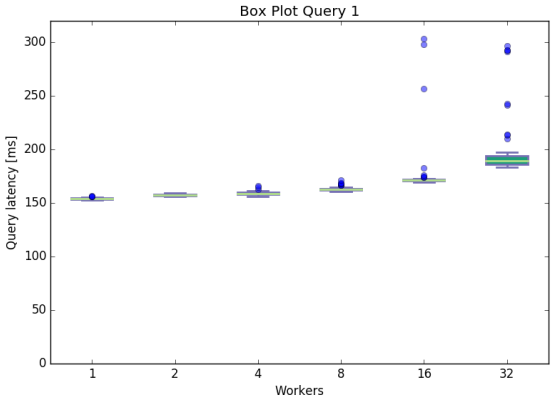

Figure 3: Large Fattree

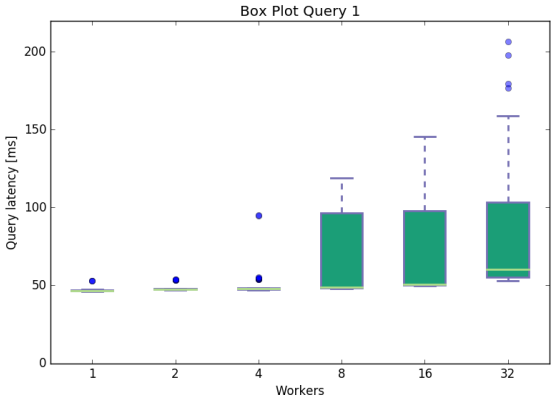

Figure 4: Small Fattree

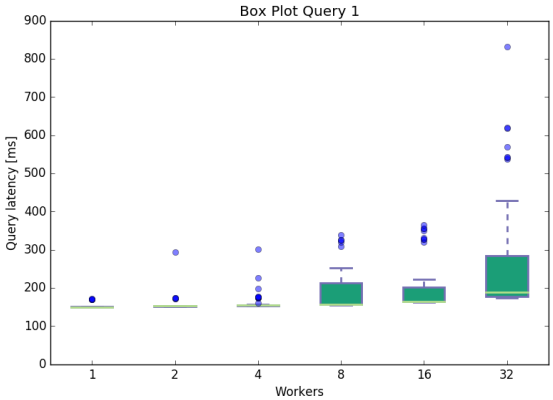

Figure 5: Large Jellyfish

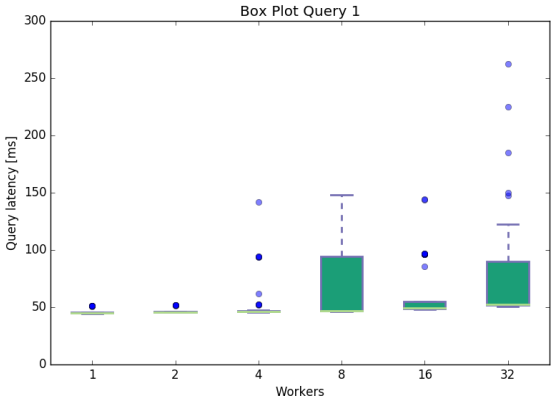

Figure 6: Small Jellyfish

6 Query2

```
SELECT n.name 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.

Dataflow

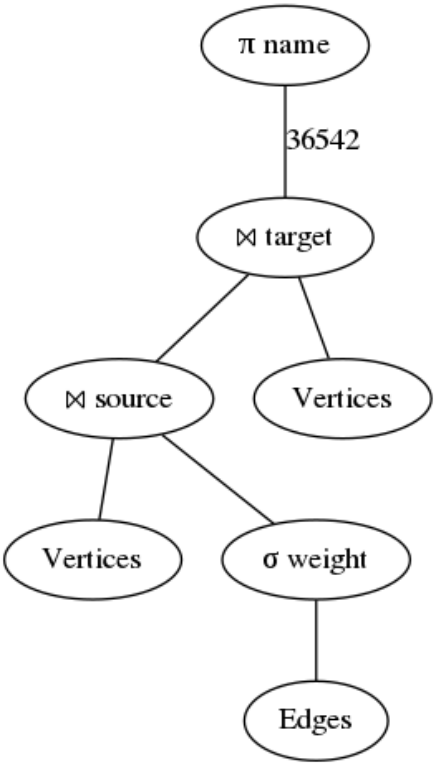

Figure 7: Dataflow Query2

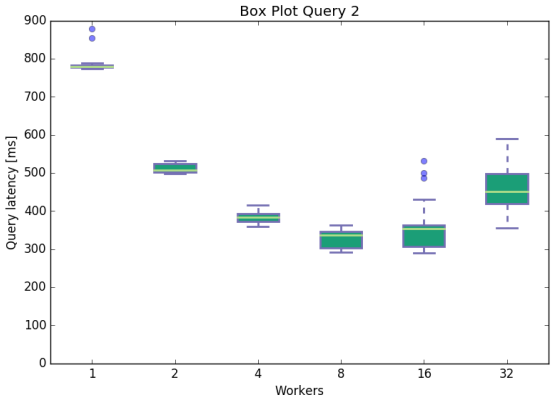

Figure 8: Large Fattree

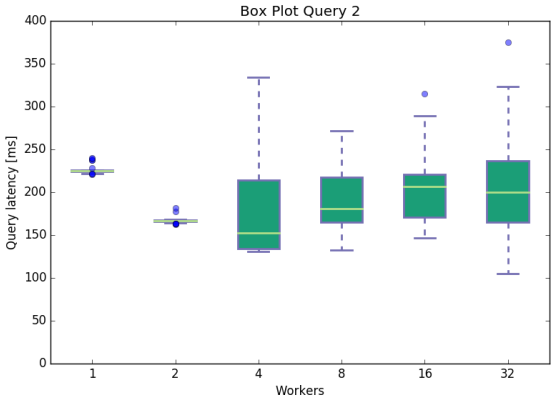

Figure 9: Small Fattree

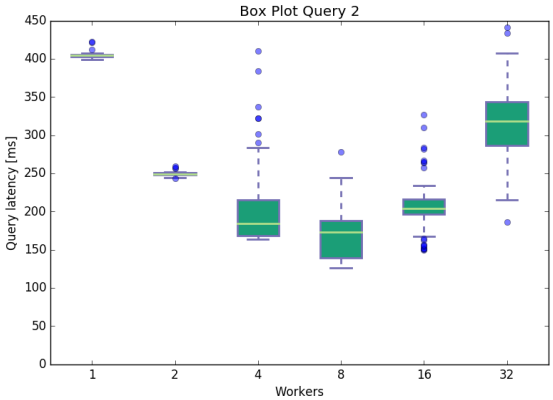

Figure 10: Large Jellyfish

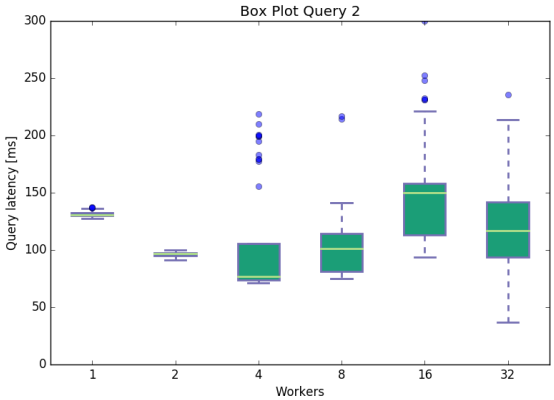

Figure 11: Small Jellyfish

7 Query3

```
SELECT n.name 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.

Dataflow

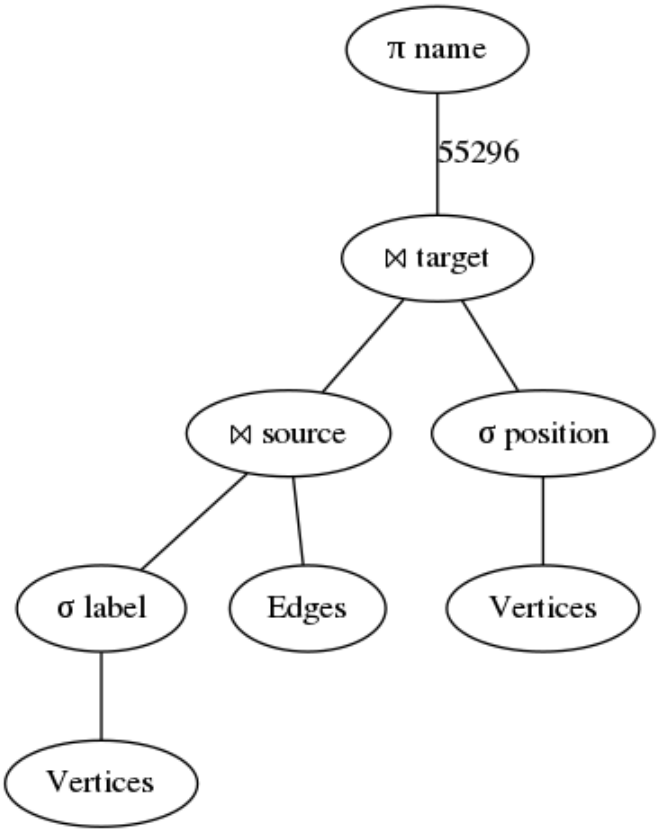

Figure 12: Dataflow Query3

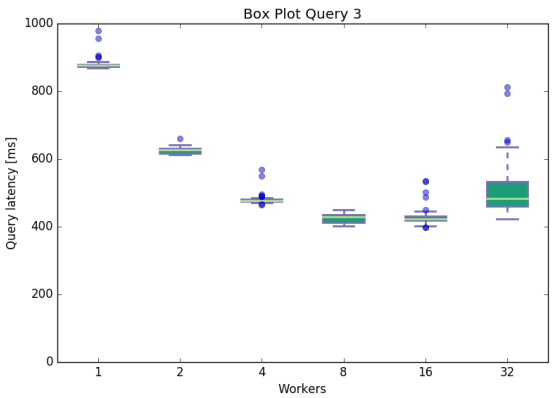

Figure 13: Large Fattree

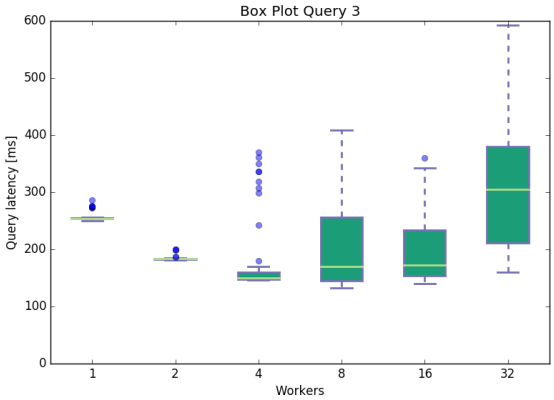

Figure 14: Small Fattree

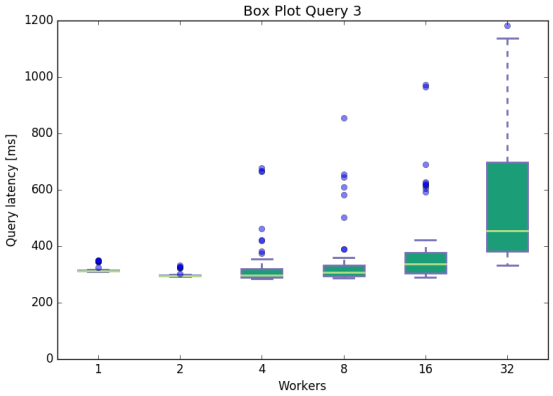

Figure 15: Large Jellyfish

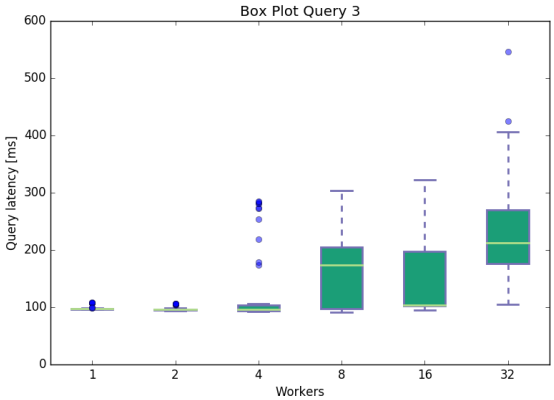

Figure 16: Small Jellyfish

8 Query4

```
SELECT n.name 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.

Dataflow

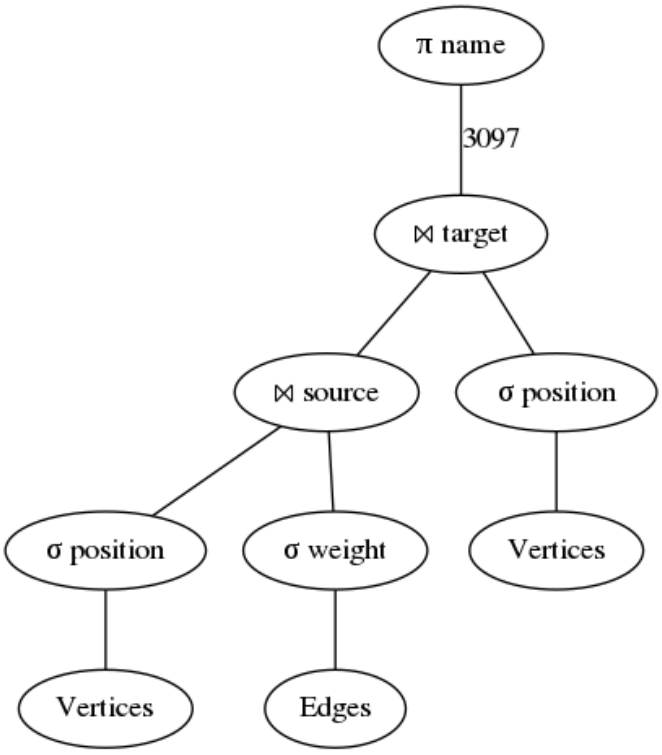

Figure 17: Dataflow Query4

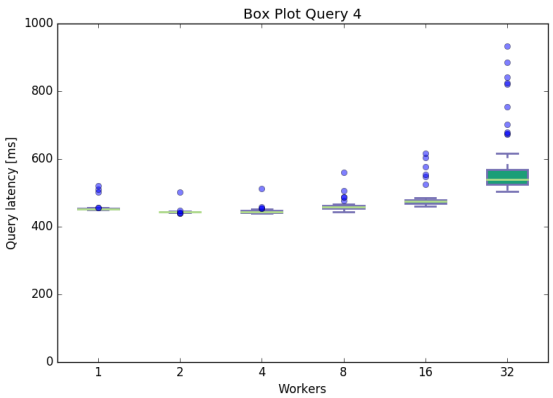

Figure 18: Large Fattree

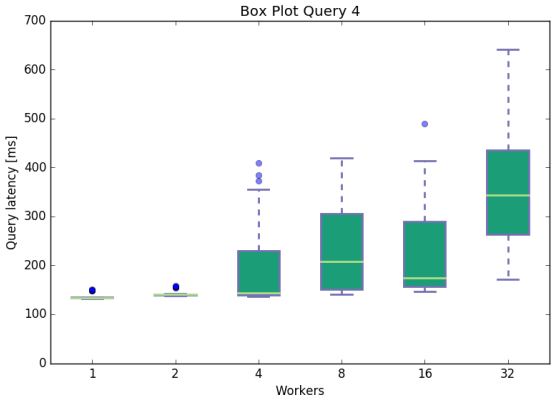

Figure 19: Small Fattree

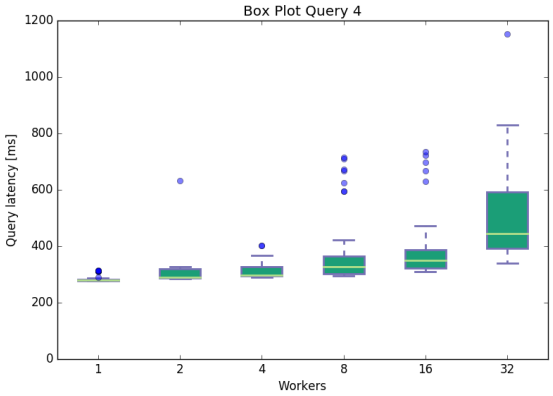

Figure 20: Large Jellyfish

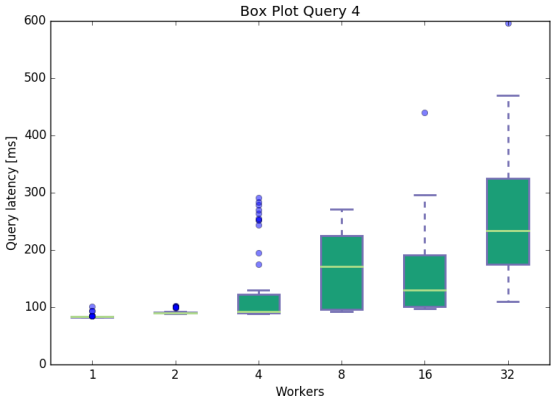

Figure 21: Small Jellyfish

9 Query5

```
SELECT v.name 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.

Dataflow

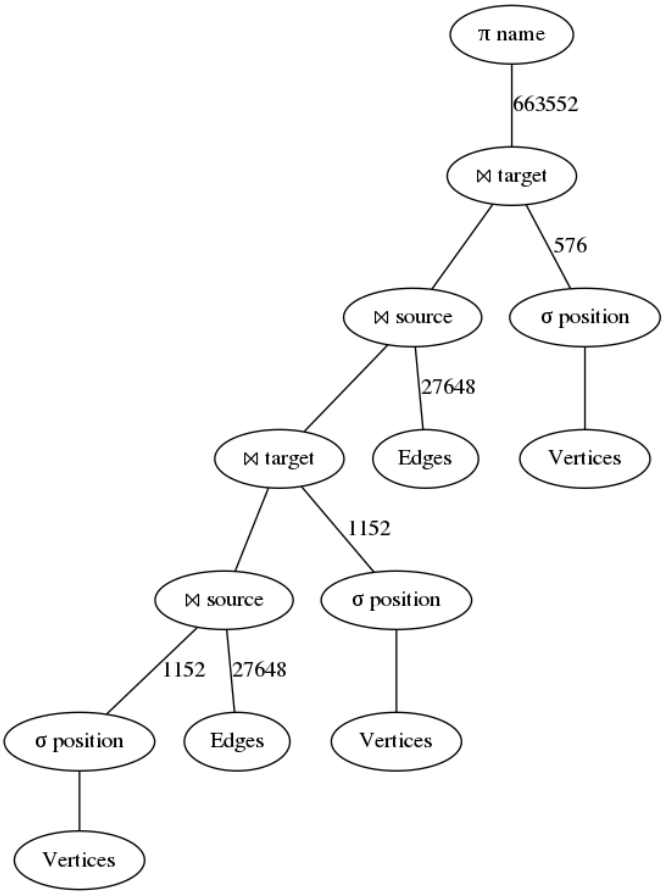

Figure 22: Dataflow Query5

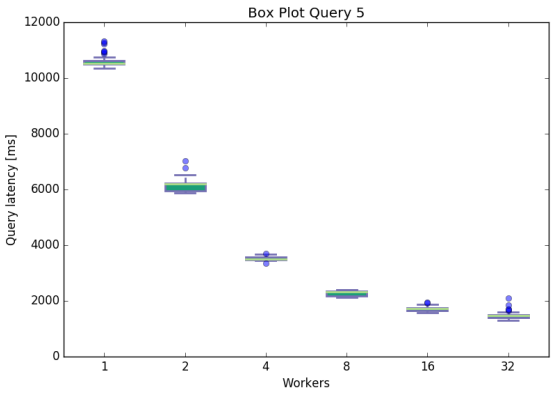

Figure 23: Large Fattree

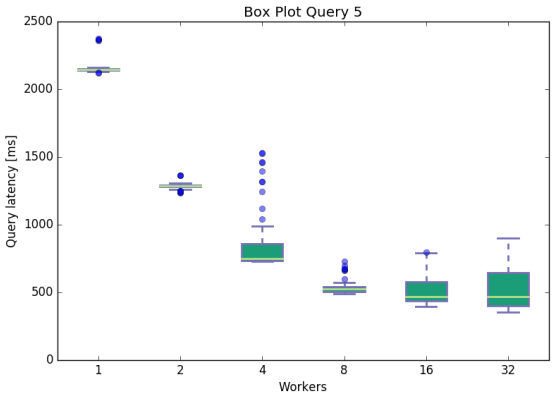

Figure 24: Small Fattree

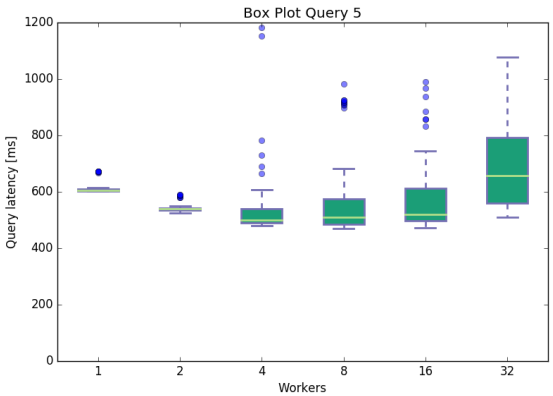

Figure 25: Large Jellyfish

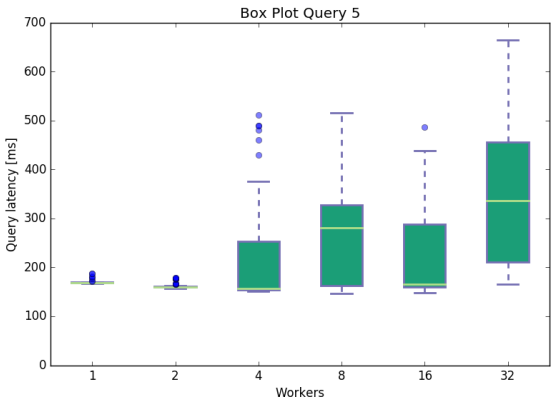

Figure 26: Small Jellyfish

10 Query6

```
SELECT v.name 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.

Dataflow

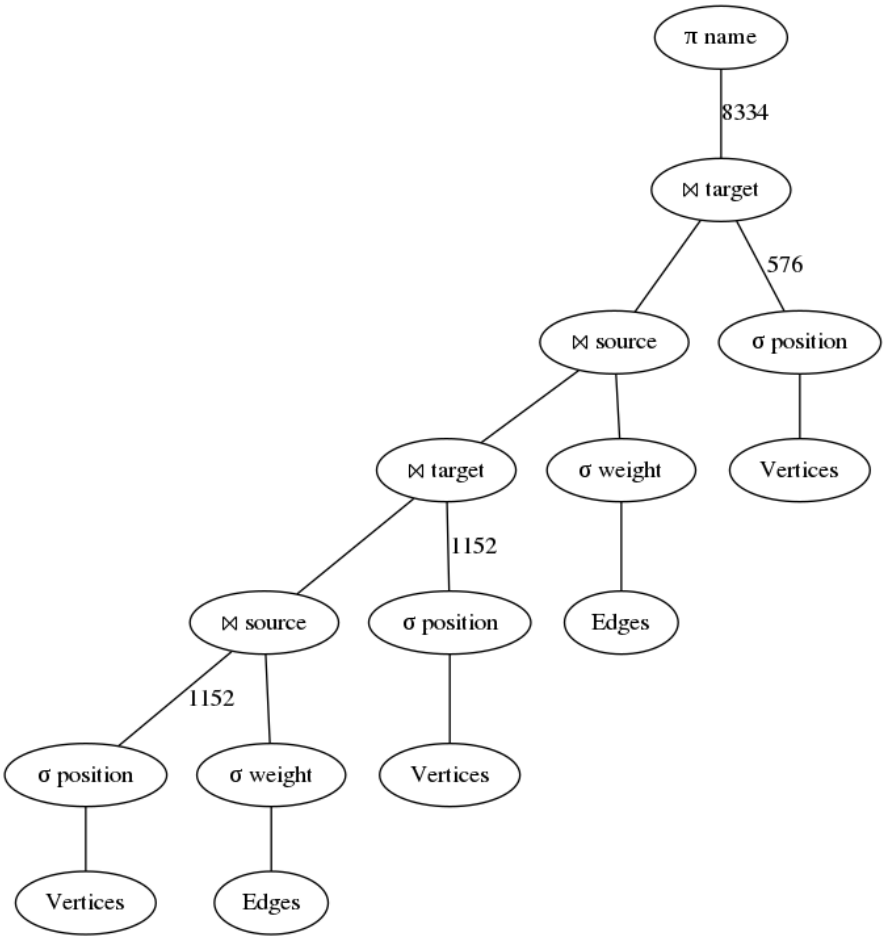

Figure 27: Dataflow Query6

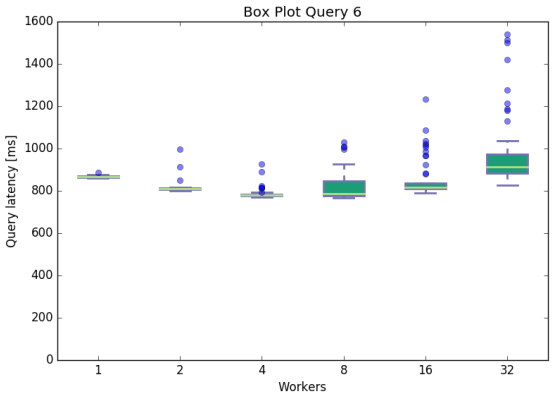

Figure 28: Large Fattree

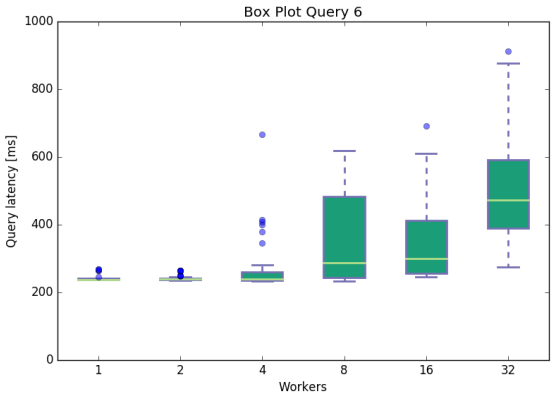

Figure 29: Small Fattree

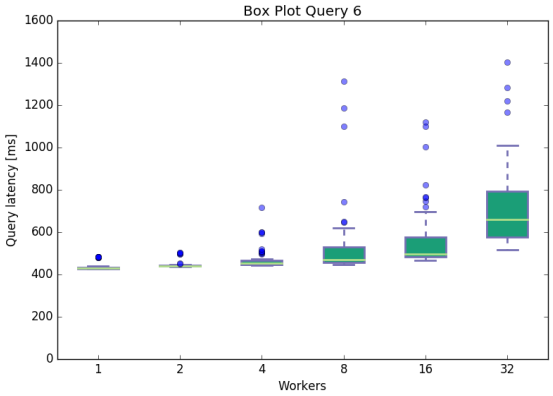

Figure 30: Large Jellyfish

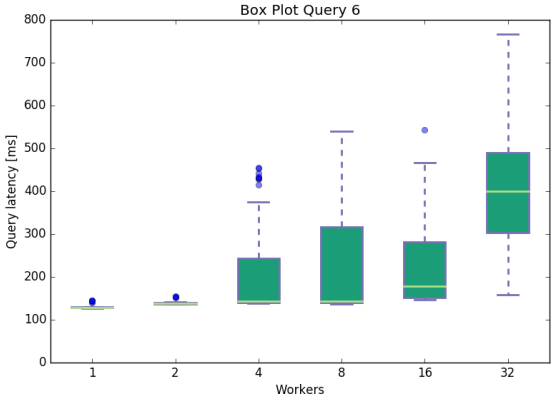

Figure 31: Small Jellyfish

11 Query7

```
SELECT v.name 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.

Dataflow

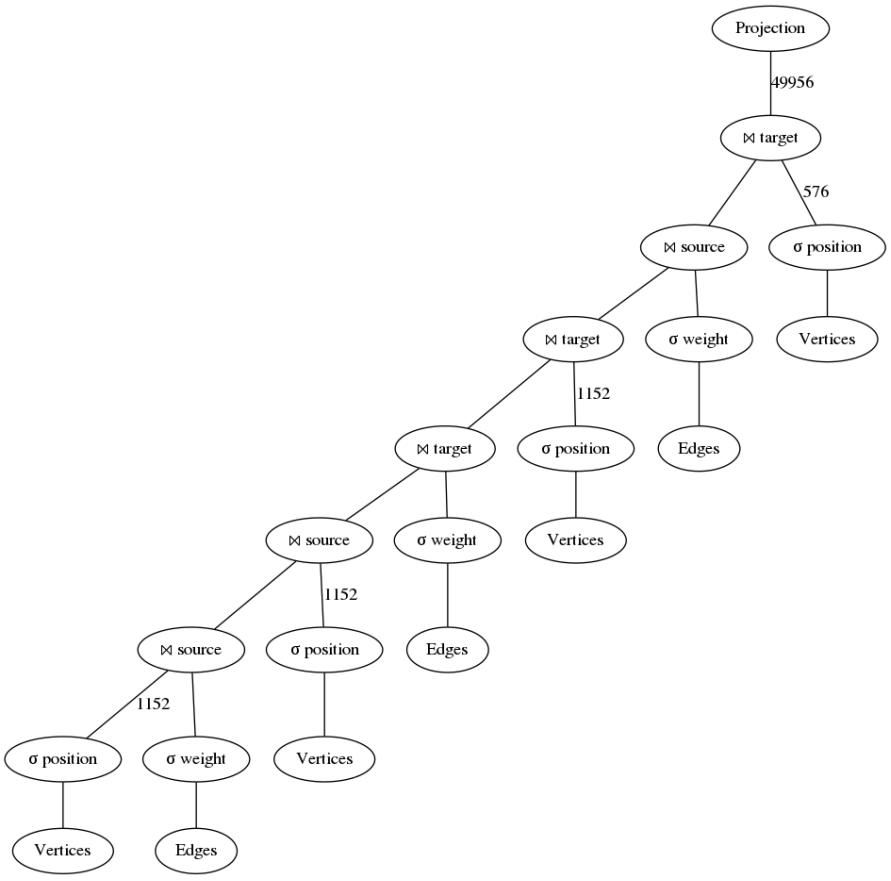

Figure 32: Dataflow Query7

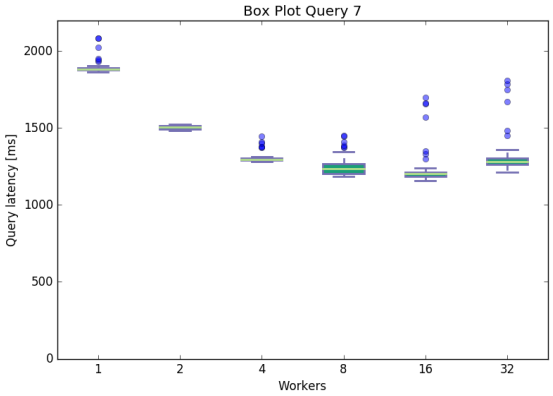

Figure 33: Large Fattree

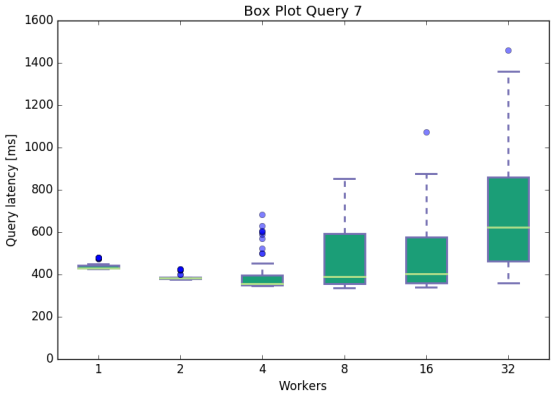

Figure 34: Small Fattree

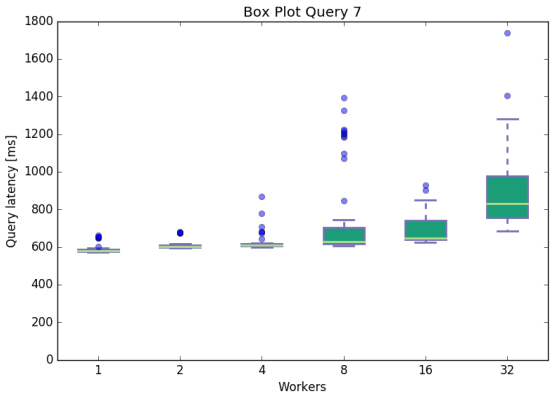

Figure 35: Large Jellyfish

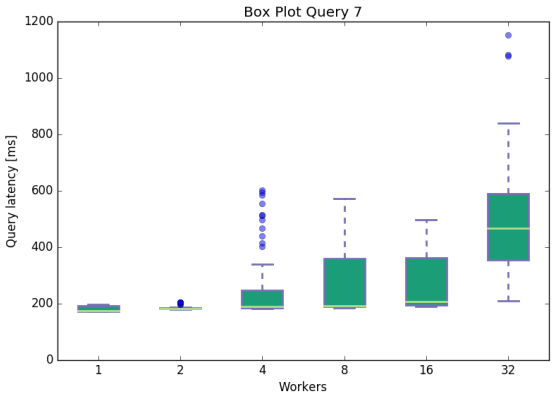

Figure 36: Small Jellyfish