Notes And Implementation Of Algorithms

Junjie Wang,Shigang Quan,Ruizhe Zhao

January 11, 2019

1 Outline & Notes

Our assigned paper is: Efficient Matching for Web-Based PublishSubscribe Systems(`https://link.springer.com/chapter/10.1007/10722620_17`)

Our mindmap of the paper is given in Figure 1, followed by some complementary explanations:

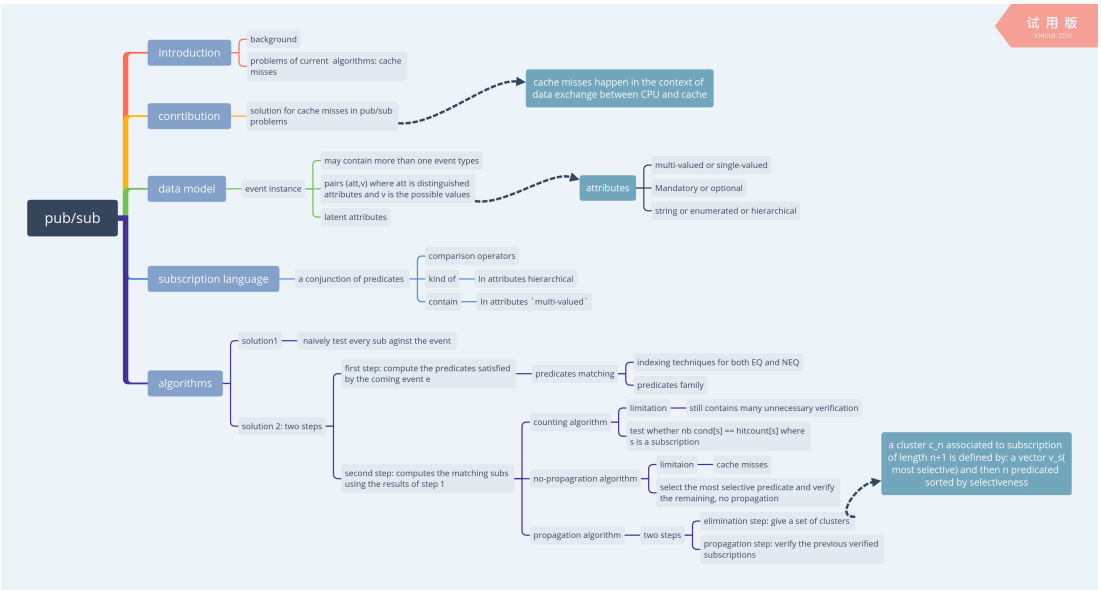

Figure 1: mindMap

1.1 Cache Misses

Cache misses problem is to some extent the background of the paper. Cache misses happen during the process of CPU's reading data from cache and when we have to frequently re-read the data since only a little part of the data previously read are useful.(e.g. in specific context, computing the transpose of a 513*513 matrix is much faster than a 512*512 one) One thing that helps to cache misses is to strengthen the locality of our program(e.g.the locality of the data and the instructions.)

As we can see later, **propagation algorithm** in this paper aims to optimize the look-up through dealing with tricky cache misses.

1.2 Data Model

In tha data model, an event instance takes the form of pairs (att,value). Attributes can be single-valued or multi-valued, optional or mandotory, enumerated or hierarchicy. There is also an eventType pair and explicit pair which contains all the attributes.

As for the subscription language, there are 5 operators, namely 3 comparators plus 'kind of' and 'contains'. In coincidence with the experiment part of the paper(also for the simplicity:)), we only take the first 3 operators into consideration.

2 Inplementation Of Algorithms

With reference to the code of opIndex(2014), we implemented the 3 algorithms in C++, and the complete code can be found on github(`https://github.com/dbgns/pub-sub`).

There are mainly two steps in matching. The first step is matching the predicates with an incoming event over all the predicates. And the second step is using the matched predicates to match subscriptions. The **second steps** algorithms are as follows:

2.1 Counting Algorithm

2.1.1 Pseudocode In Paper

In figure 2 is the pseudocode of counting algorithm in the paper. It iterates the matching predicates first and increment the hitcount by 1 if a predicate of a sub is satisfied(this part is achieved by hashing signature in C++ followed)

inputs :
nb cond[]: nb cond[] is vector of integer of size $| S |$
pred to subs[]: is an association table of size $| P |$
MatchingPreds[]: output of predicate matching algorithm
body:

1 $matched \leftarrow \{\}$
2 $hitcount \leftarrow | S | \#0$
3 **foreach** $p \in MatchingPreds$
4 **foreach** $s \in pred\ to\ subs[p]$
5 $hitcount[s] \leftarrow hitcount[s] + 1$
6 **for** $i = 1$ **to** $| S |$
7 **if** $hitcount[s] == nb\ cond[s]$ **then** $matched \leftarrow matched \cup \{s\}$ **endif**
8 return matched

Figure 2: Counting Algorithm

2.1.2 C++ Implementation

We first initialize the signature using the matchedPreds(results of the first step) and then increment hitcount if signature is satifsfied.

```
for(int j=0;j<subList.size();j++)
{
   for(int k=0;k<subList[j].size;k++)
   {
      if(sig[hashSignature(subList[j].constraints[k].att,
           subList[j].constraints[k].op,subList[j].constraints[k].value)])
      {
         if(++hitCount[subList[j].id] == counter[subList[j].id])
            {matchedSubs++;break;}
      }
   }
}
```

2.2 Propagation Algorithm

2.2.1 Pseudocode In Paper

In figure 3 is the pseudocode of propagation algorithm given in the paper. It first create clusters according to the matched predicates in first step(a bit like classification) and then iterate the predicates sorted by selectiveness(We only pass the subs verified by

previous predicates down because they are more selective, and that's where the name of the algorithms comes from).

Cluster propagation
inputs
MatchingPreds: output of predicate matching algorithm
$c = (v_s, v_1, \cdots, v_n)$: a cluster of subscriptions of size n
preds bitmap: bitmap containing $\mid P \mid$ bits. *preds bitmap*$[p] = 1$
if predicate p is verified 0 otherwise
initialmatch: indices of selected subscriptions in v_s
body:

```
1  match ← initialmatch
2  for i = 1 to n {
3     new ← ∅
4     foreach j ∈ match {
5        if preds bitmap[v_i[j]] then new ← new ∪ j }
6     match ← new }
7  matched ← v_s[match]
8  return matched
```

Figure 3: Propagation Algorithm

2.2.2 C++ Implementation

The process of initializing vectors v[1]... v[n] is emitted here for simplicity and can be found in the complete code. As follows: the outside loop is the clusters and in inner loops we pass down the subs.

```
for(int clu=0;clu<3*MAX_ATTS*MAX_VALS;clu++)
{
   if(!sig[clu] || vs[clu].size() == 0) continue;

   //matched results
   vector<int> matched;
   for(int i=0;i<vs[clu].size();i++)
   matched.push_back(i);
   if(matched.size()==0) continue;

   for(int i=0;i<vs[clu][0].size-1;i++)
   {
```

```
      vector<int> newMatch;
      for(int j=0;j<matched.size();j++)
      if(satisfyCnt(v[clu][i][j],pub)) newMatch.push_back(j);
      if(newMatch.size()!=0) matched = newMatch;
      else   { matched.clear(); break; }
   }
   matchedSubs += matched.size();
}
```

2.2.3 Non-Propagation Algorithm

Non-Propagation Algorithm is only a little different from the propagation algorithm mentioned above. It also creates clusters first but it will test all the remaining predicates besides the most selective one rather than propagate down.

2.2.4 C++ Implementation

```
for(int clu=0;clu<3*MAX_ATTS*MAX_VALS;clu++)
{
   if(!sig[clu] || vs[clu].size() == 0) continue;
   //matched results
   for(int i=0;i<vs[clu][0].size-1;i++) //namely the number of predicates
   for(int j=0;j<v[clu][i].size();j++)
   if(satisfyCnt(v[clu][i][j],pub))
   {
      //use 1 rather than 0 since the most selective predicate is already
          verified
      if(--counter[vs[clu][j].id] == 1) {++matchedSubs;break;}
   }
}
```

3 Our Experiment Results

We ran our experiments on i7CPU 2.8GHZ having Linux as its operating system. The parameters and ranges are the same with the original authors(given in figure 4).

Detailed graphs are as follows: In figure 5 where the size of subscriptions(S) is varying, the performance of the non-prop algorithm is very close to the counting algorithm. In figure 6 where V is varying, we can see that with V increasing, the time decreses(though there are some noise).

Name	Description	Range
S	number of subscriptions	15000 to 3000000
A	number of attributes per event	1 to 16
V	number of possible values per attribute	10 to 200
NP	number of predicates per subscription	1 to 16
EQ	number of equality predicates per subscription	0 to 16
NEQ	number of nonequality predicates per subscription	0 to 16

Figure 4: Propagation Algorithm

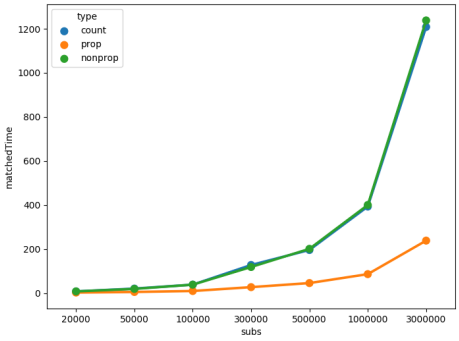

Figure 5: NP=EQ=4,V=35,S is varying

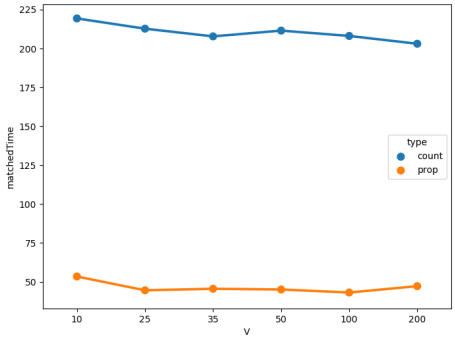

Figure 6: S=500K,NP=EQ=4,V is varying

In figure 7 where NP(the number of predicates) increases, we can see that the time increases. In figure 8, with EQ(NP keeps the same) increases, time decreases,**indicating that equality is more selective than the nonequality and also takes less time to index.**

4 Comparison Of The algorithms

From the graphs above and the organization of the 3 algorithms, we can conclude that:

1. Propagation algorithm outperforms the other 2 algoirthms in all the contexts shown above.
2. Although better than naively test the event e again each subscription, there are also many unnecessary tests in the counting algorithms, as it doesn't considert the relationship between different predicates.

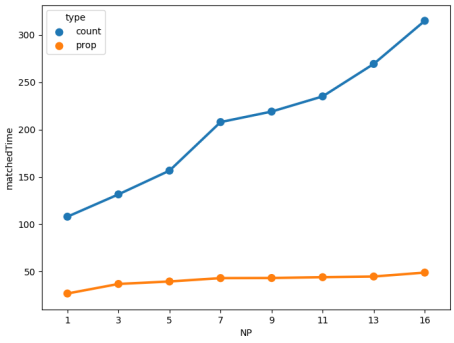

Figure 7: V=35,S=500K,NP is varying

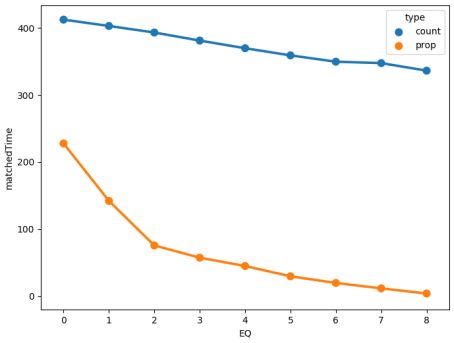

Figure 8: V=35,S=500K,EQ is varying

3. Easy to see, it's unnecessary to verify the predicates if previous predicates, which are more selective, are not satisfied. Therefore, there is a waste of time in the non-propagation algorithm. But as the number of subs increases, the non-propagation algorithm outperforms the counting algorithm(may be for the sake of clusters)

4. Propagation algorithm handles cache misses problem as the propagation ensures the locality of data(only subs verified by previous steps will be passed down and subs grouped by the clusters will have higher probability to match on the same event.)

5 Limitation & Things We Thought But Did Not Try

1. The other two operators,namely 'kind of' and 'contains'. We did not consider the two ,as they will bring in much difficulty in the generation of attributes and values.

2. Other data structure. We mainly use hashing and binary search in the C++ implementation. However, there are many places we can still optimize(e.g. the sorting algorithm). Also, to better handle the cache misses problem, we can use data structure like CSS-tree. We may try them later.

3. Sad to say, our experimental envrionment is much better than the orginal authors, but our algorithms ran much slower(inefficient implementaion).... :(

References

[1] Joo Pereira et al. *Efficient Matching for Web-Based Publish/Subscribe Systems*, 2000.

[2] Dongxiang Zhang et al. *An efficient publish/subscribe index for e-commerce databases*, 2014.