**Deep Bodra**

**5801 1841**

**DBI Week 8(3/9 - 3/13)**

**Detailed explanation of the topics covered in class**

**3/10/2020**

1. Generalized Linear Aggregates (GLA’s)
   1. They are a far more generalized version of the aggregates that allows us to compute user defined aggregates, pass one aggregate as a parameter to another aggregate.
   2. Mathematical Model
      1. Notation
         1. S => set of states
         2. s ∊ S
         3. t => a tuple
      2. Properties
         1. s+t -> s’ [Adding tuple to a state produces another state]
            1. Not commutative
         2. s⊕s’ ->s’’ [2 states can be added to produce 3rd state]
            1. Associative
            2. Commutative
         3. s ≡s’ means the states are statistically equivalent. This does not mean that they produce the same result. In the context of sampling, it means that the states have similar distribution
   3. Formal Model
      1. Init() -> initialize state
      2. AddItem() -> s+t
      3. AddState() -> s⊕s’
      4. Finalise() -> generate output either trivially or by some computation
2. Aggregates using GLA’s

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Aggregates | Average | TopK | Bundle  (Merge a set of GLA’s so that they can be passed as parameters to other GLA’s) | GroupBy | LinearRegressor |
| State/Init | sum=0  count=0 | Set of k tuples  Φ | Set of states G1, .....Gn | Hash  Empty hash | s=[Φ]nxn  count=0 |
| s.AddItem(t) | sum+=t.A  count++ | If len(s)<K:  S+t  Else:  If t is worst in s:  Ignore t  Else:  1. Remove worst from s  2. add t | Gi.AddItem(t) | g=EnsureGroup(h(t))  g.AddItem(t) | s+=(t.xT)(t.x)  count++ |
| s.AddState(o) | sum+=o.sum  count+=o.count | For t in o.s:  AddItem(t) | Gi.AddState(o) | For k, g in o.hash  g’=EnsureGroup(k)  g’.AddState(g) | s+=o.s  count+=o.count |
| Finalise() | sum/count | Return all the tuples in the final state | Concatenate tuples | For k, g in hash  Concatenate( k, g) and apply aggregate function | Solve equations |

* 1. TopK
     1. Use of heap for TopK will not lead to bad performance as case 2 is taken all the time(Why? Based on the probability theory, the new tuple that comes will be worse than the current set of tuples as the current set of tuples gets better every iteration)
     2. The branch will be predicted correctly most of the time and there is no decrease in performance because of random access of heaps
  2. GroupBy
     1. Merge has bad parallelism
     2. It has long tail problem that is as the group grows bigger the merge time increases and the load will be on just one thread
     3. Spoiler: Segmentor GLA will solve this problem
  3. Other GLA’s: Generalized Linear Regressor, Clustering, etc

**3/12/2020**

1. Histogram GLA
   1. Collaboration with Prof. Ahmed Helmy to find network user profile for each user in the network
   2. This is based on the Netflow and DHCP mapping tables
   3. Query:
      1. Join Netflow and DHCP mapping tables
      2. GroupBy MacAddress
      3. Computer HistogramGLA
   4. New model to solve this
      1. Multi iterator model
      2. GetIterator(): manages concurrency on the common hash
      3. GetNext(): gets the next item for the iterator
2. Segmented GLA
   1. It solves the tail end problem of GLA
   2. Idea: If tuples t1, t2 are to be in same bucket( i.e. their hashes are same) then they are put in same segment of the GLA
   3. How to achieve this?
      1. Use same hash function for segmentation and groupby
   4. Merge: This is trivial, thread obtains lock and simply puts its records.
   5. As soon as hash is built the merge phase starts without having to wait for other threads to finish building the hash.
   6. This is because no two threads have records that evaluate to the same hash. [ensured by the common hash function we chose]

--THANK YOU--