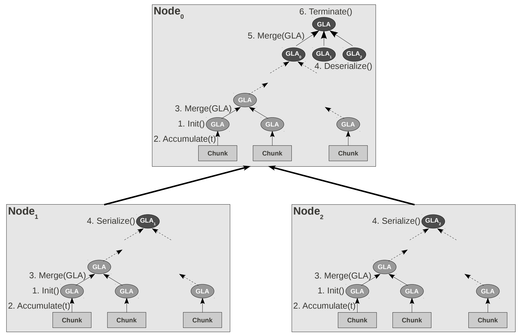
GLADE: Big Data Analytics Made Easy

Yu Cheng, Chengjie Qin, Florin Rusu

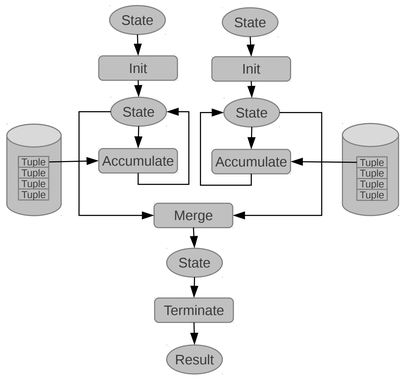
Electrical Engineering and Computer Science, University of California, Merced
ycheng4@ucmerced.edu, cqin3@ucmerced.edu, frusu@ucmerced.edu

Architecture

- GLADE is a scalable and **efficient** parallel framework for Big Data analytics



- Generalized Linear Aggregates (GLA)
  UDA interface: ***Init***, ***Accumulate***, ***Merge***, and ***Terminate***
  Users implement any aggregate computation as GLA



Data & Tasks

```
CREATE TABLE UserVisits (
 sourceIP VARCHAR(16),
 destURL VARCHAR(100),
 visitDate DATE,
 adRevenue FLOAT,
 userAgent VARCHAR(64),
 countryCode VARCHAR(3),
 languageCode VARCHAR(6),
 searchWord VARCHAR(32),
 duration INT );
```

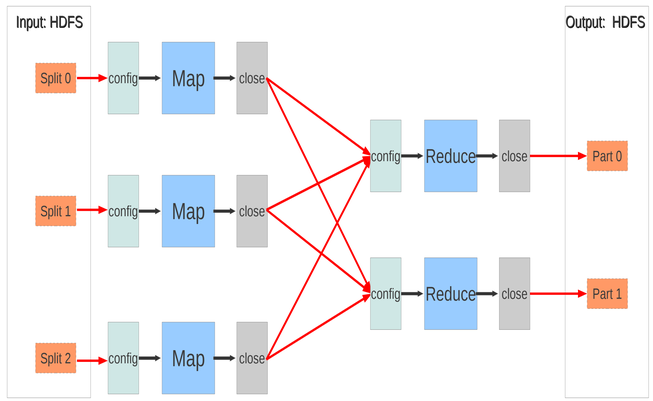
- **Average**: average time a user spends on a page
- **Group By**: ad revenue generated by a user across all the visited web pages
- **Top-K**: users who generated the largest one hundred ad revenues on a single visit
- **K-Means**: five most representative ad revenues

GLADE GLAs

- **Average**: sum; count
  Accumulate(Tuple) – add tuple value to sum; increment count
  Merge(GLA) – add sums and counts
  Terminate() – calculate average as sum divided by count
- **Group By**: hash table (user → revenue)
  Accumulate(Tuple) – if find user, add revenue; otherwise insert new user
  Merge(GLA) – merge hash tables; for same user, sum the revenues
- **Top-K**: min-heap with K entries
  Accumulate(Tuple) – if heap top smaller than tuple, extract top and insert new tuple; reorganize heap
  Merge(GLA) – call Accumulate for each element in GLA argument
- **K-Means**: K centers; K Average GLAs
  Accumulate(Tuple) – find closest center and call Accumulate on corresponding Average GLA
  Merge(GLA) – call Merge on corresponding Average GLAs
  Terminate() – compute new centers

Hadoop Map-Reduce

- Java & Pig Latin



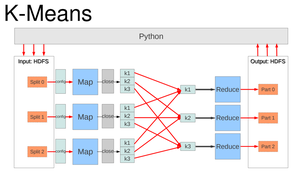
▸ Average and Top-K

Calculate local result and emit with single pre-defined key in *close*.

▸ Group By

Map-Reduce direct application.

▸ K-Means



Python as external driver to control iterations. Separate key for each center.

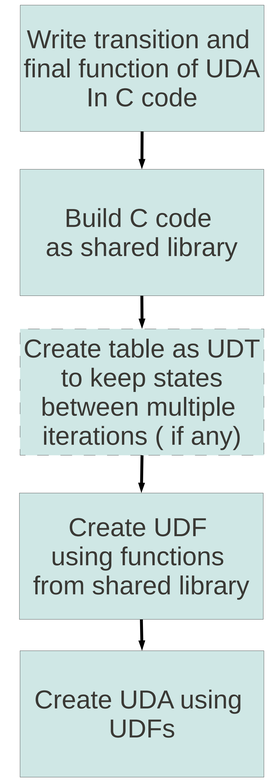
PostgreSQL UDAs

- **SQL definition**

```
CREATE AGGREGATE kmeans (
  double precision,
  double precision[])
(
 STYPE = double precision[],
 SFUNC = kmeans_transit,
 FINALFUNC = kmeans_final,
 INITCOND = '{0}'
);
```

- **C implementation**

```
PG_FUNCTION_INFO_V1(kmeans);
Datum kmeans(PG_FUNCTION_ARGS)
{
        ArrayType *wparray = (ArrayType *) PG_GETARG_RAW_VARLENA_P(0);
        double *data;
        int wpLen = my_parse_array_no_copy((struct varlena*) wparray,
                sizeof(float8), (char **) &data);
        if( wpLen == 1){
                //use arg[2] to retrieve serialized centers from previous iteration
                ArrayType *initwparray = (ArrayType *) PG_GETARG_RAW_VARLENA_P(2);
                double *initwp;
                int initwpLen = my_parse_array_no_copy((struct varlena*) initwparray,
                        sizeof(float8), (char **) &initwp);
                wparray = my_construct_array(initwpLen + 2*M, sizeof(float8), FLOAT8OID);
                wpLen = my_parse_array_no_copy((struct varlena *)wparray,
                        sizeof(float8), (char **)&data);
                memcpy(data, initwp, initwpLen * sizeof(float8));
                int i;
                for( i = M; i < 2*M; i ++)
                        data[i]=0;
        }
        float8 next_attr = PG_GETARG_FLOAT8(1);
        int i, flag;
        flag = 0;
        for( i = 1; i < M ; i ++ )
        {
                if( fabs(data[i] - next_attr) < fabs(data[flag] - next_attr) )
                        flag = i;
        }
        data[flag + M ] = data[flag+M] + next_attr;
        data[flag + 2 * M ] = data[flag + 2 * M] + 1;
        PG_RETURN_ARRAYTYPE_P(wparray);
}
```



Experimental Results

- **single node, 20GB**

Task	*Execution Time (seconds)* GLADE	PostgreSQL
Average	3	79
Group By	34	5,895
Top-K	5	96
K-Means (one iteration)	4	62

- **8 nodes, 20GB per node**

Task	*Execution Time (seconds)* GLADE	Hadoop
Average	3	1,261
Group By	60	1,875
Top-K	5	1,260
K-Means (one iteration)	4	1,297