GLADE: A Highly-Scalable Architecture-Independent Framework for Efficient Analytics

FLORIN RUSU
Email: frusu@ucmerced.edu Web: http://faculty.ucmerced.edu/frusu

1. Natural to specify and implement. The main idea is to provide a very clean abstraction to the user with little or no micro-architecture specific code.
2. Declarative with respect to parallelism/distribution. The user should not be forced with any decisions on parallel execution regarding load balancing, amount of parallelism, communication between various groups of nodes, etc.
3. Independence of data representations and I/O. The user should not be burdened with details of how data is made available (files, databases, etc.) and how data is represented. From user’s point of view the data should look like simple tuples but the framework should have the freedom to select more suitable internal representations.
4. Allow natural non-transactional behavior. A major critique of relational databases is that they are too rigid. Everything must be a triple or a relation. As long as GLAD is not coupled to back-end data sources, it can co-exist with a multitude of data sources.
5. More hand-crafted code performance. Specifically, we want the GLAs to allow very efficient computation comparable with the best code that can be written. Ideally, no performance penalty is incurred by this abstraction.

Abstract Data Type (ADT) State
An implicit assumption is that the state is manageable in size. An explicit assumption is that the part of the state that is retrieveable is finite. This is crucial since we want to restrict the user to the set of re-translation primitives.

Core API
- **Initialize**: This acts as a constructor and initializes the state to reflect the fact that no data has been incorporated.
- **AddItem(T elem)**: A function that updates the state with the data in one tuple. This function has to be commutative. In other words, the same state should be obtained irrespective of the order of incorporating multiple tuples.
- **AddItem(GLA other)**: A function that incorporates the other state into the current state. The operation is required to be commutative and associative. This is essential for the partitioning of the state.

Migration API
- **PartitionGLA**: Where the state of the GLA can be transformed into a compact linear representation.
- **serialize**: Serializes the state of the GLA into a byte representation.

Result Extraction API
The GLAs are allowed to have only one extraction API the user can call to get the final result. If the GLA is used in contexts where a single value is needed, the method **Finalize** is defined. If multiple tuples are allowed - e.g., when the GLA is piped into another operator, an iterative iteration is supported.

Formalization
\[(GLA) = (D, S, F, +, \alpha, \delta)\]
- \(D\) is the universe of tuples and defines the type of the triple the GLA can process.
- \(S\) is a set of possible states. Intuitively the states are data that the GLA has to maintain.
- \(F\) is a function from states to states, that function allows extraction of results from the GLA.
- \(S = S_0\) is an initial state. \(S_0\) is the element with respect to \(\delta\).
- \(\alpha\) and \(\delta\) are commutative and associative. The commutativity and associativity of the two operations is crucial for the correctness of the system.

The connection between the formalization and the GLA is:
\[S_0 = \text{the state created by Initialize}\]
\[\text{is implicitly passed to AddItem}\]
\[^1\] is the state of the final GLA

Higher-Order Functions
GLA Contraction: Given \(S_0\), \(\alpha\), or \(\delta\), a GLA type, the contraction \(S_0\) \(\alpha\) \(\delta\) is a GLA that has as state the contraction of the states of the parts - the \(\alpha\) and \(\delta\) operations the recursive application of the corresponding \(\alpha\) and \(\delta\) for each sub GLA. It can be easily seen that the resulting contraction is indeed a GLA that has the same reductions and is commutative and associative.

Filter-GLA: Given a GLA \(G\) and a filtering condition \(\phi\), a new GLA \(\phi G\) that incorporates only tuples that satisfy the condition into the GLAs in each operation + for the new GLA \(G\) the \(\phi\) \(\phi\) \(\phi\) is true and needs to be added. It is easy to see that this results in a commutative +. Operation \(\phi\) is unchanged.

Transform to GLA: A similar idea is to allow arbitrary transformations of the tuple before passing it on to the GLA. As long as the transformation is a deterministic function from tuples of triple (with side effects or memory), the user operation +, that applies the transformation and then calls the GLA on the transformed tuples, is commutative.

Join-GLA: Given a GLA \(G\) and a tuple set \(T\), a new GLA \(\overline{G} T\) that works on tuples formed by joining tuples from \(T\) are obtained by defining the plus + as the joining operation followed by \(\overline{G}\). \(\overline{G}\) is defined as the transitive closure of the GLA and the function +.

Generic `GroupBy` BY: Given a grouping function \(g(t)\) and a GLA \(G\) a `GroupBy(G, BY)` GLA can be obtained by serializing the GLA from groups and scanning all GLAs of type \(\alpha\) operation - strict deterministic function of tuple grouping using \(g(t)\) and looks for the corresponding GLA \(G_\phi\). If no such GLA exists, it creates one using \(\alpha\) and \(\delta\) functions.

Top-K: The number of top-m GLAs of type \(\alpha\) operation + and operation \(\delta\). Operation \(\delta\) returns the \(m\) group of the second `GroupBy` BY GLA while looking up the corresponding group in the first GLA. The two top-Ks are merged using these two operations.

---

### Code Snippet

```java
template<class ElemType, class ResultType>
class GLA_average {
public: 
  void initialize() { ... }
  void finalize() { ... }
  // this is the state data for the GLA
  class GLA_average{...};
  // Performance
  ResultType result()
  ResultType done(); 
  // Returns
  GLA_average(); 
  GLA_average(const GLA_average &); 
  void add(const ElemType & val);
  void addSum(const ElemType & val, const ReferenceType & ref);
};

// AddItem(GLA_average other)
void AddSum(GLA_average other) {
  count += 1;
  sum += other sum;
}
```

### GLA – Generalized Linear Aggregates

### GLA – GLA Distributed Engine

### Experimental Results

#### Data: 155 million tuples, 20GB

#### Top-K: 5

#### K-Means: calculates the five most representative (5 centers) advertisement revenues

#### Group By: computes the ad revenue generated by a user across all the visited web pages

#### Group By

#### Average:

#### K-Means Average Time per Iteration

#### K-Means Average Time per Iteration

---

**Server:** A AID Optimus server at 20GB, 32GB memory; 76 hard-disks connected through 5 RAID controllers for a total maximum bandwidth of 4GB/s. Unix (Uname 2.6.32-6) fast ethernet (4GB) connected to other machines by a 100MBs. (Linux 2.6.32-6) kernel version 2.6.32-26; one instance of ShareFS (2GB) is loaded on each disk for a total maximum of 1.3TB.

**Cluster:** 76 nodes, A AID Optimus server at 2GB, 4GB memory and single data with a bandwidth of 100MB/s. (Linux 2.6.32-6) kernel version 2.6.32-6) on each node, coordinator rate on a separate node.