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

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

GLA – Generalized Linear Aggregates

Requirements

1. *Natural to specify and implement.* The main idea is to provide a very clean abstraction to the user with little or no unnecessary details.
2. *Declarative with respect to parallelism/distribution.* The user should not be burdened with any decisions on parallel execution, including synchronization, amount of parallelism, communication between various nodes.
3. *Independence of data representation and I/O.* The user should not be burdened with details of how data is made available (disk, memory, cache, 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 representations.
4. *Allow natural non-relational behavior.* A major critique of relational databases is that they are too rigid. Everything must be a tuple or a relation. As long as GLAs are not required to feed data into other operators, it is desirable to allow them to produce non-relational results.
5. *Have hand-written 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 writable is *private*. This is crucial since we want to unburden the user from the use of synchronization primitives.

Core API

- `Initialize`. This acts as a constructor and initializes the state to reflect the fact that no data has been incorporated.
- `AddItem(Tuple t)`. 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 the tuples.
- `AddState(GLA other)`. A function that incorporates the `other` state into the current state. The `other` GLA is destroyed at the end of the operation. The operation has to be equivalent to performing `AddItem` for all the tuples incorporated in `other` state. The operation is required to be commutative and associative. This is effectively a way to merge two GLAs into one.

Migration API

- `Serialize` specifies how the state of the GLA can be transformed into a compact binary representation.
- `Deserialize` specifies how to re-construct the GLA from the binary representation.

Result Extraction API

The GLAs are allowed to have **any** extraction API if the user code is the final destination. If the GLAs are used in contexts where a single value is needed, the method `Finalize` is defined. If multiple tuples are allowed, e.g., when the GLA result is piped into other operators, an iterator interface is supported.

Formalization

$$(\text{GLA}) = (\mathcal{D}, \mathcal{S}, \mathcal{F}, +, \oplus, S_0)$$

- $\mathcal{D}$ is the universe of tuples and defines the type of tuple the GLA can *process*.
- $\mathcal{S}$ is a set of possible states; informally, the states are data that the GLA has to maintain.
- $\mathcal{F}$ is a function from states to results; this function allows extraction of results from the GLA.
- $+ : \mathcal{S} \star \mathcal{D} \to \mathcal{S}$ allows tuples to be incorporated into state.
- $\oplus : \mathcal{S} \star \mathcal{S} \to \mathcal{S}$ allows combination of states.
- S_0 is the null element with respect to $\oplus$.
- $+$ and $\oplus$ 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 API is:

- S_0 is the state created by `Initialize`
- $+$ is implemented by `AddItem`
- $\oplus$ is implemented by `AddState`
- $\mathcal{F}(S)$ corresponds to `Finalize`

Higher-Order Functions

GLA Concatenation: Given $\mathcal{G}_1, \ldots, \mathcal{G}_n$, n GLA types, the concatenation $\mathcal{G}_1 || \ldots || \mathcal{G}_n$ is a GLA that has as state the concatenation of the states of the n parts, the $+$ and $\oplus$ operations the successive application of the corresponding $+$s and $\oplus$s for each GLA. It can be easily seen that the resulting concatenation is indeed a GLA since the resulting $+$ and $\oplus$ are commutative and associative.

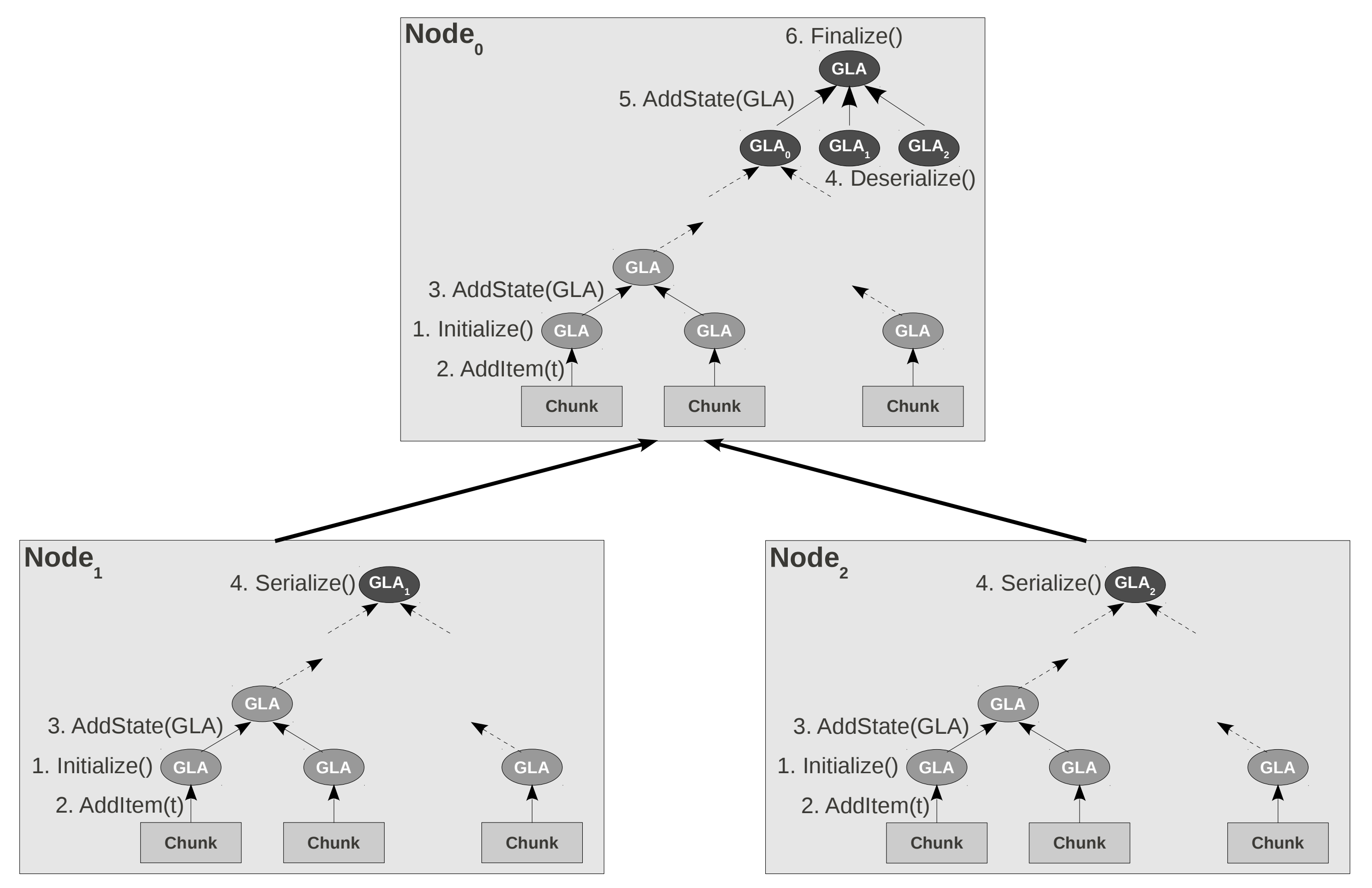
Filter+GLA: Given a GLA $\mathcal{G}$ and a filtering condition $\sigma(t)$, a new GLA $\mathcal{G}'$ that incorporates only tuples that satisfy the predicate σ. Operation $+$ for the new GLA performs the old $+$ if $\sigma(t)$ is true and nothing otherwise. It is easy to see that this results in a commutative $+$. Operation $\oplus$ is unchanged.

Transform+GLA: A similar idea to filtering is to allow arbitrary transformations of the tuple before passing it as an argument to the $+$ function. As long as the transformation is a deterministic function from tuples to sets of tuples (without side effects or memory), the new operation $+$, that first applies the transformation and then calls the old $+$ for each resulting tuple, is commutative.

Join+GLA: Given a GLA $\mathcal{G}$ and a tuple set $\mathcal{T}$, a new GLA $\mathcal{G}'$ that works on tuples formed by *joining* input tuples with tuples from $\mathcal{T}$ can be obtained by defining the new $+$ as the joining operation followed by the old $+$. Operation $\oplus$ remains the same. The hash table that allows the efficient joining can be sent as the constant state to the GLA. Since tuples can be joined independently, the new $+$ is commutative.

Generic `GROUP BY`: Given a grouping function $g(t)$ and a GLA $\mathcal{G}$, a `GROUP BY` GLA can be obtained by setting the state as a mapping between groups and states of GLAs of type $\mathcal{G}$. Operation $+$ first determines the group using $g(t)$ and looks for the corresponding GLA $\mathcal{G}_{g(t)}$. If no such GLA exists, it creates one (using S_0 of $\mathcal{G}$). The state of the new or existing GLA is updated using its $+$ operation and tuple t. Operation $\oplus$ traverses the groups of the second `GROUP BY` GLA while looking up for the corresponding group in the first GLA. The two sub-GLAs are merged using their $\oplus$ operation.

GLADE – GLA Distributed Engine



DataPath

1. When a new query arrives in the system, the code to be executed is first generated, then compiled and loaded into the execution engine. Essentially, the waypoints are configured with the code to execute for the new query.
2. Once the storage manager starts to produce chunks for the new query, they are routed to waypoints based on the query execution plan.
3. If there are available work units in the system, a chunk and the task selected by its current waypoint are sent to a work unit for execution.
4. When the work unit finishes a task, it is returned to the pool and the chunk is routed to the next waypoint.

Shared-Memory

1. When a chunk needs to be processed, the GLA Waypoint extracts a GLA state from the list and passes it together with the chunk to a work unit. The task executed by the work unit is to call *AddItem* for each tuple such that the GLA is updated with all the tuples in the chunk. If no GLA state is passed with the task, a new GLA is created and initialized (*Initialize*) inside the task, such that a GLA is always sent back to the GLA Waypoint.
2. When all the chunks are processed, the list of GLA states has to be merged. Notice that the maximum number of GLA states that can be created is bounded by the number of work units in the system. The merging of two GLA states is done by another task that calls *AddState* on the two states.
3. In the end, *Finalize* is called on the last state inside another task submitted to a work unit.

Shared-Nothing

1. The coordinator generates the code to be executed at each waypoint in the DataPath execution plan. A single execution plan is used for all the workers.
2. The coordinator creates an aggregation tree connecting all the workers. The tree is used for in-network aggregation of the GLAs.
3. The execution plan, the code, and the aggregation tree information are broadcasted to all the workers.
4. Once the worker configures itself with the execution plan and loads the code, it starts to compute the GLA for its local data. This happens exactly in the same manner as for GLADE Shared-Memory.
5. When a worker completes the computation of the local GLA, it first communicates this to the coordinator—the coordinator uses this information to monitor how the execution evolves. If the worker is a leaf, it sends the serialized GLA to its parent in the aggregation tree immediately.
6. A non-leaf node has one more step to execute. It needs to aggregate the local GLA with the GLAs of its children. For this, it first deserializes the external GLAs and then executes another round of *AddState* functions. In the end, it sends the combined GLA to the parent.
7. The worker at the root of the aggregation tree calls the function *Finalize* before sending the final GLA to the coordinator who passes it further to the client who sent the job.

Code Snippet

```
template<class ElemType, class ResultType>
class GLA_Average :  public GLA {
private:
  // this is the state data for the GLA
  unsigned long count;
  ResultType sum;
  ResultType result;
  bool done;
public:
  GLA_Average() :  count(0), sum(0),
   result(0), done(false) { }
  GLA_Average(unsigned long _count,
   ResultType _sum) :
   count(_count), sum(_sum),
   result(0), done(false) { }
  void AddItem(ElemType& val) {
   count += 1;
   sum += val;
  }
  void AddState(GLA_Average& other) {
   count += other.count;
   sum += other.sum;
  }
  void Finalize() {
   if (count > 0) {
    result = sum / count;
   }
   done = false;
  }
  bool GetNext(ResultType& where) {
   if (done == false) {
    where = result;
    done = true;
    return true;
   }
   else {
    return false;
   }
  }
  ARCHIVER_SIMPLE_DEFINITION()
};
```

Experimental Results

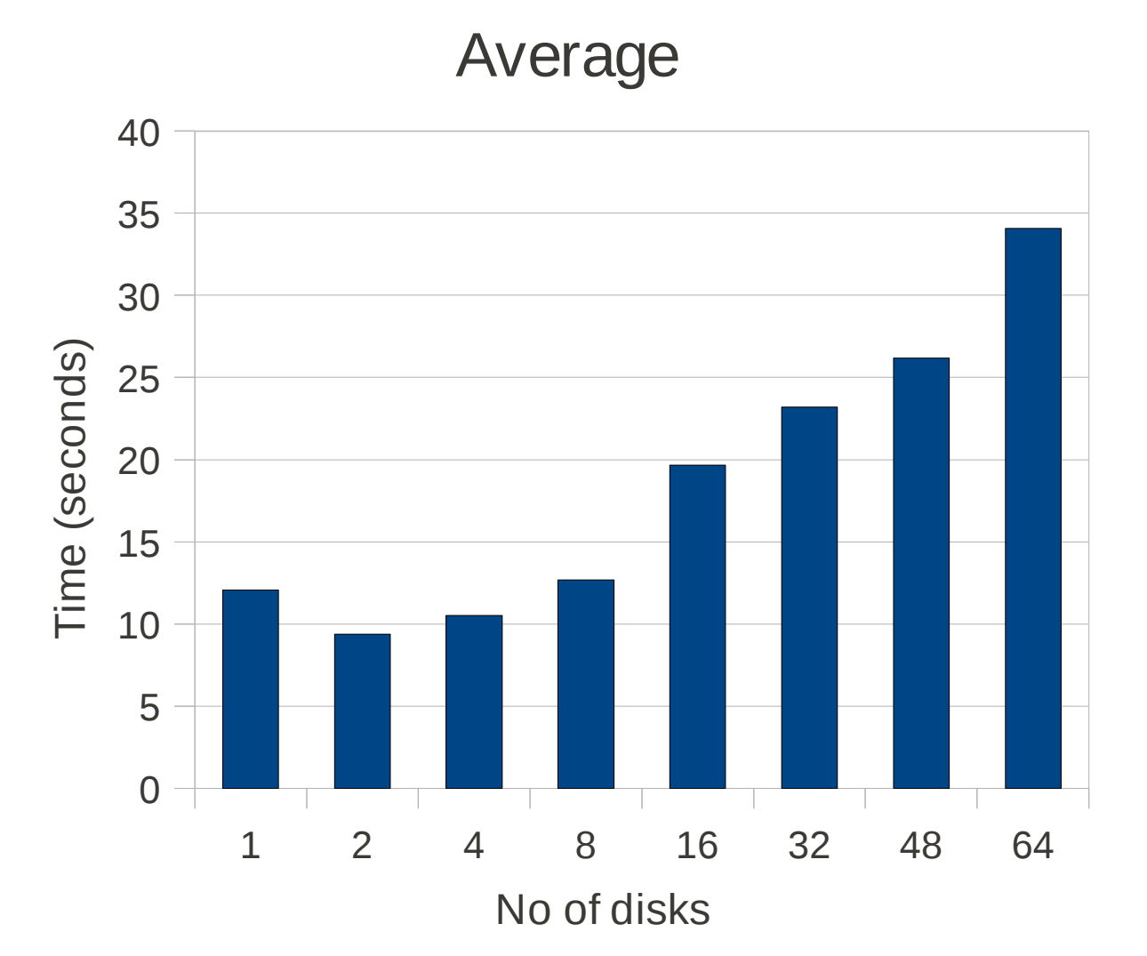
Data: 155 million tuples, 20GB

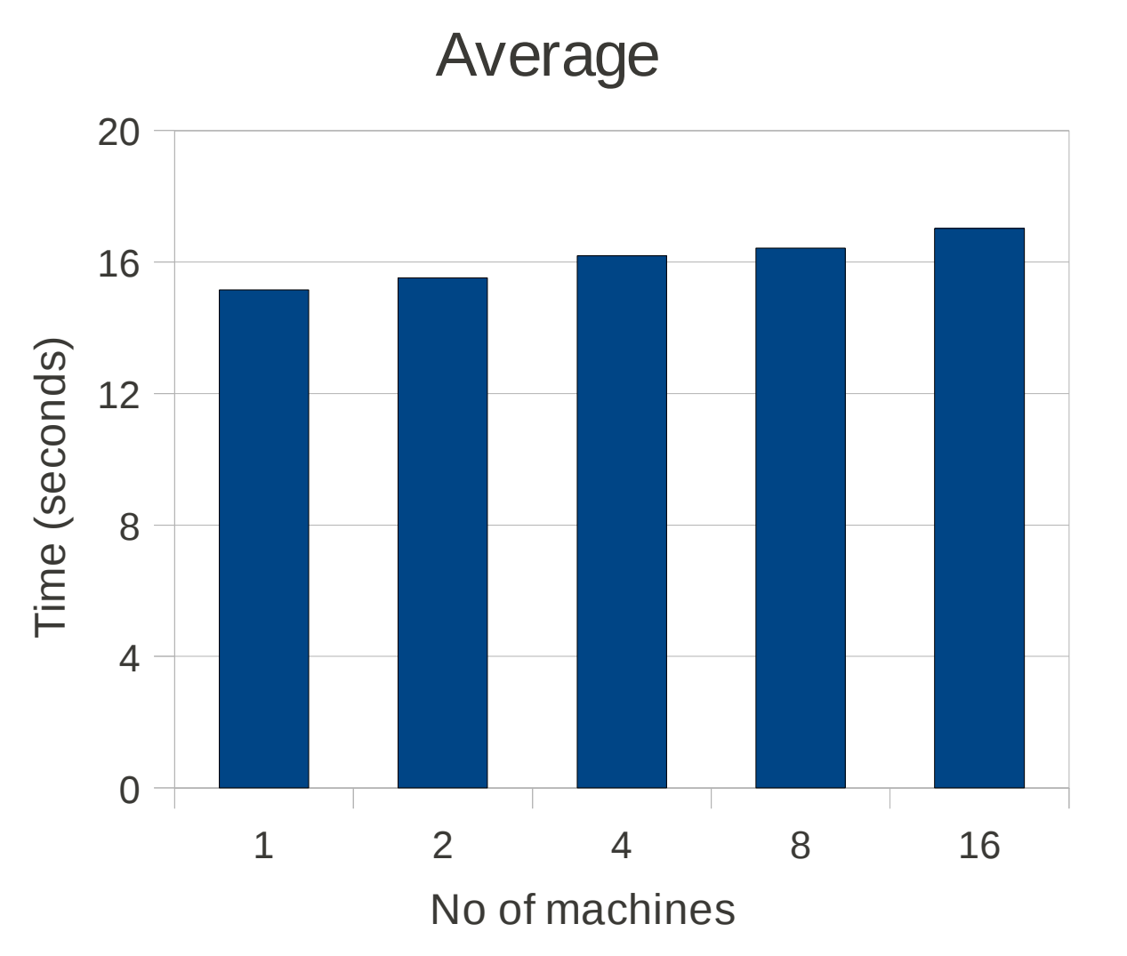
```
CREATE TABLE UserVisits (
 sourceIP VARCHAR(16),
 destURL VARCHAR(100),
 visitDate DATE,
 adRevenue FLOAT,
 userAgent VARCHAR(64),
 countryCode VARCHAR(3),
 languageCode VARCHAR(6),
 searchWord VARCHAR(32),
 duration INT );
```

Server: 48 AMD Opteron cores at 1.9GHz; 256GB memory; 76 hard-disks connected through 3 RAID controllers for a total maximum bandwidth of 3GB/s; Ubuntu 10.04 SMP Server with kernel version 2.6.32-26; one instance of `UserVisits` (20GB) is loaded on each disk for a total maximum of 1.3TB.

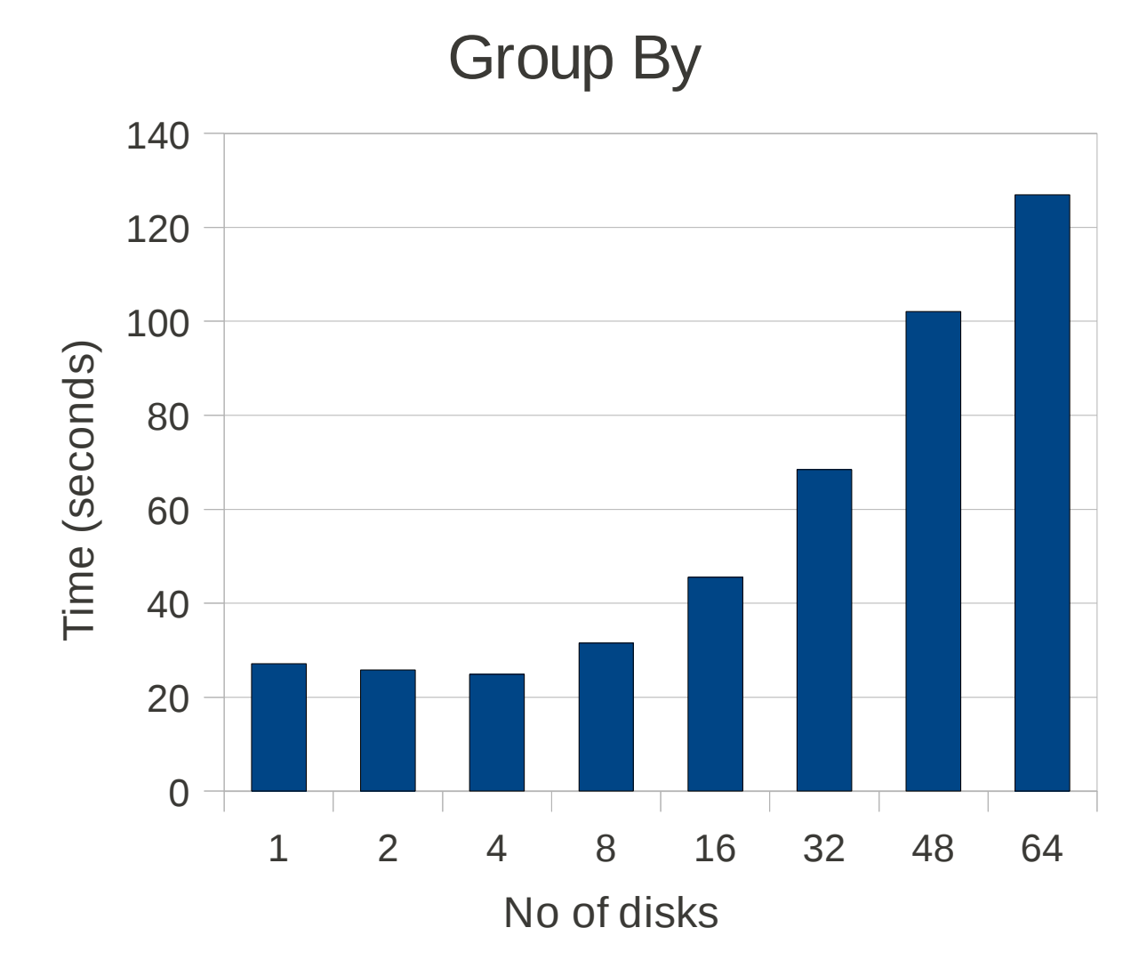
Cluster: 17 nodes; 4 AMD Opteron cores at 2.4GHz; 4GB memory; single disk with a bandwidth of 50MB/s; Ubuntu 7.4 Server with kernel version 2.6.20-16; one instance of `UserVisits` (20GB) on each node; coordinator runs on a separate node.

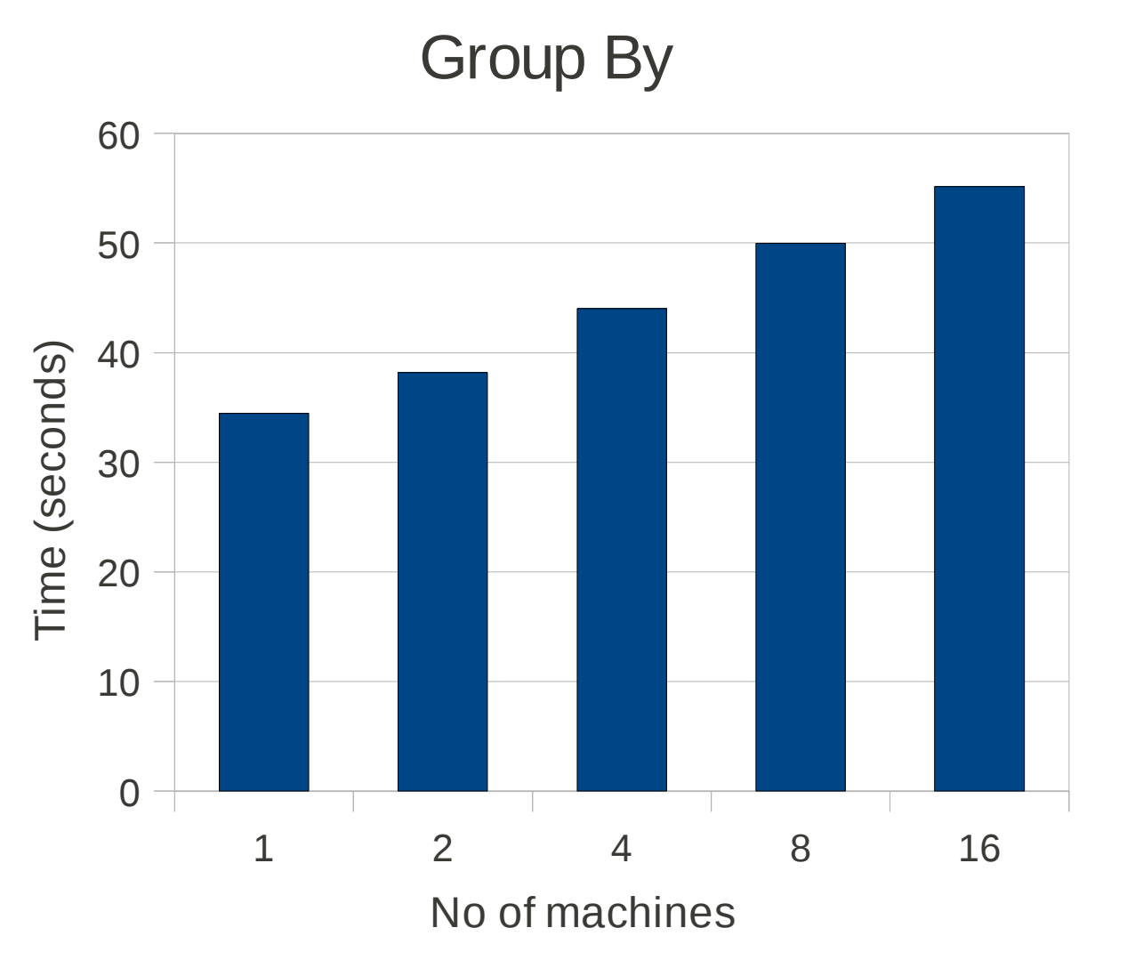
Average: computes the average time a user spends on a web page



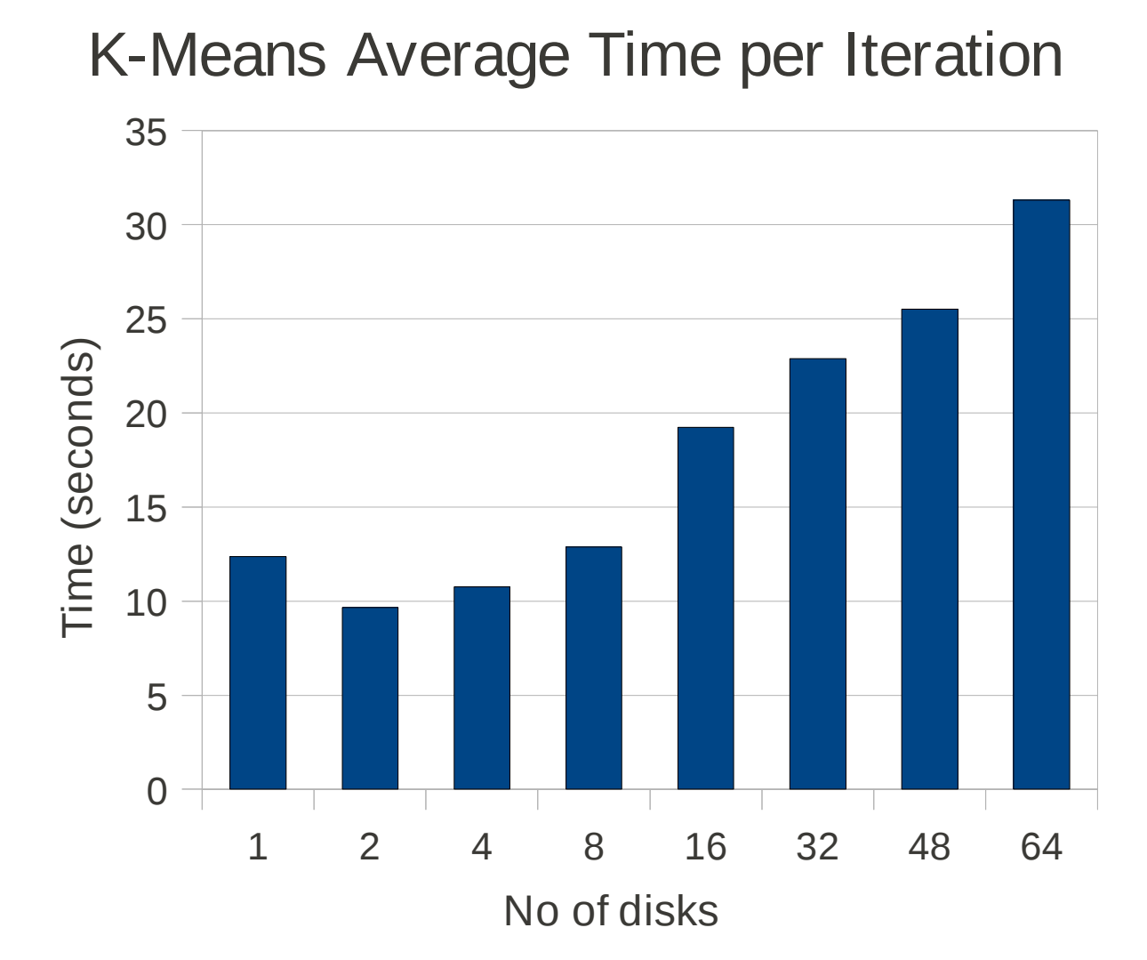


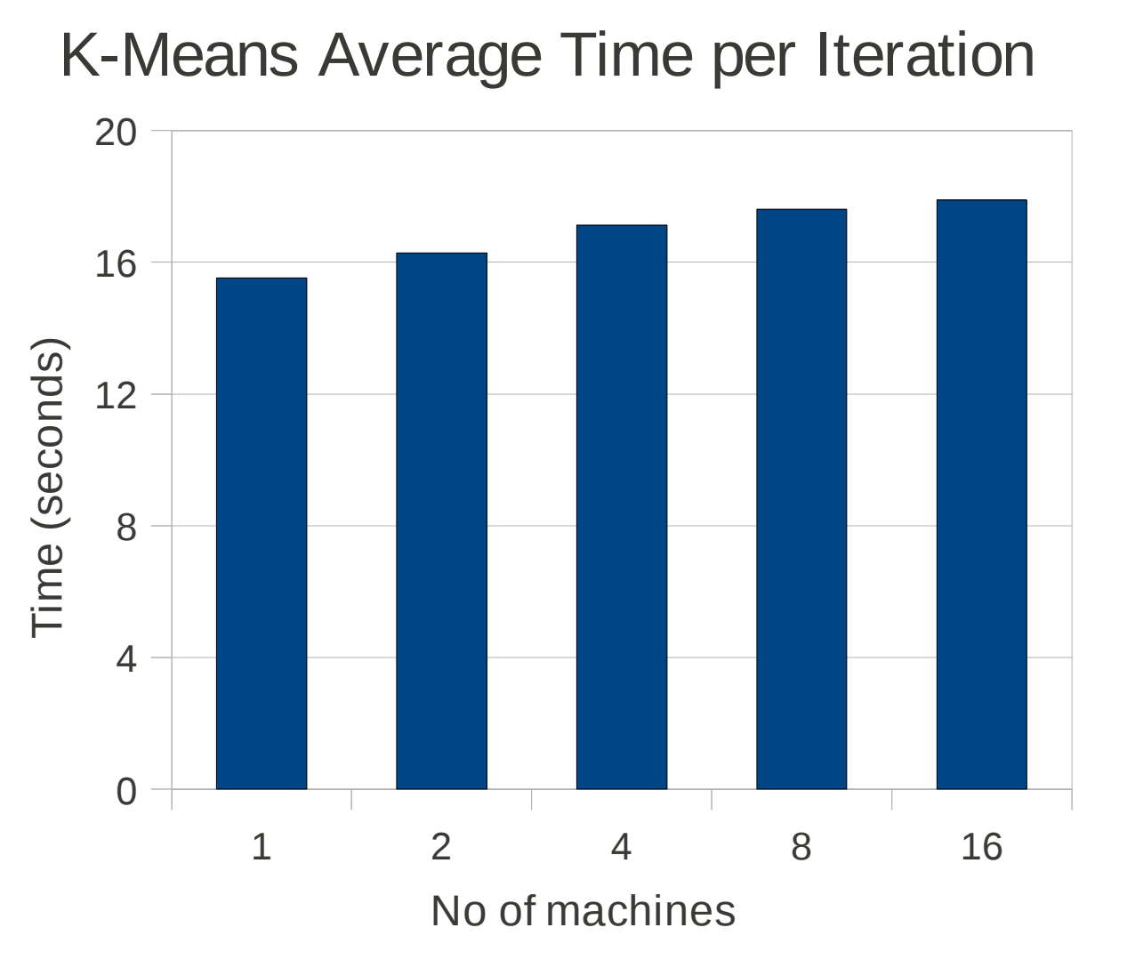
Group By: computes the ad revenue generated by a user across all the visited web pages



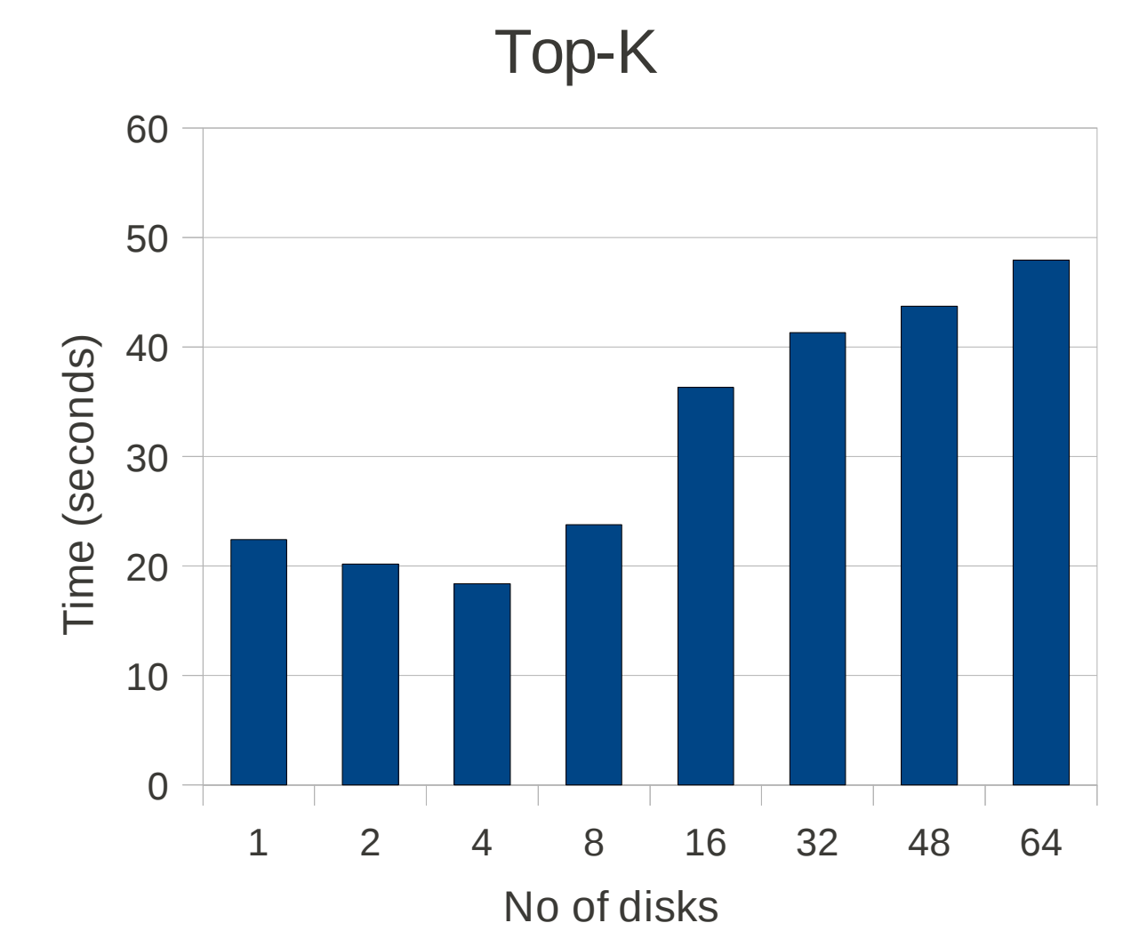


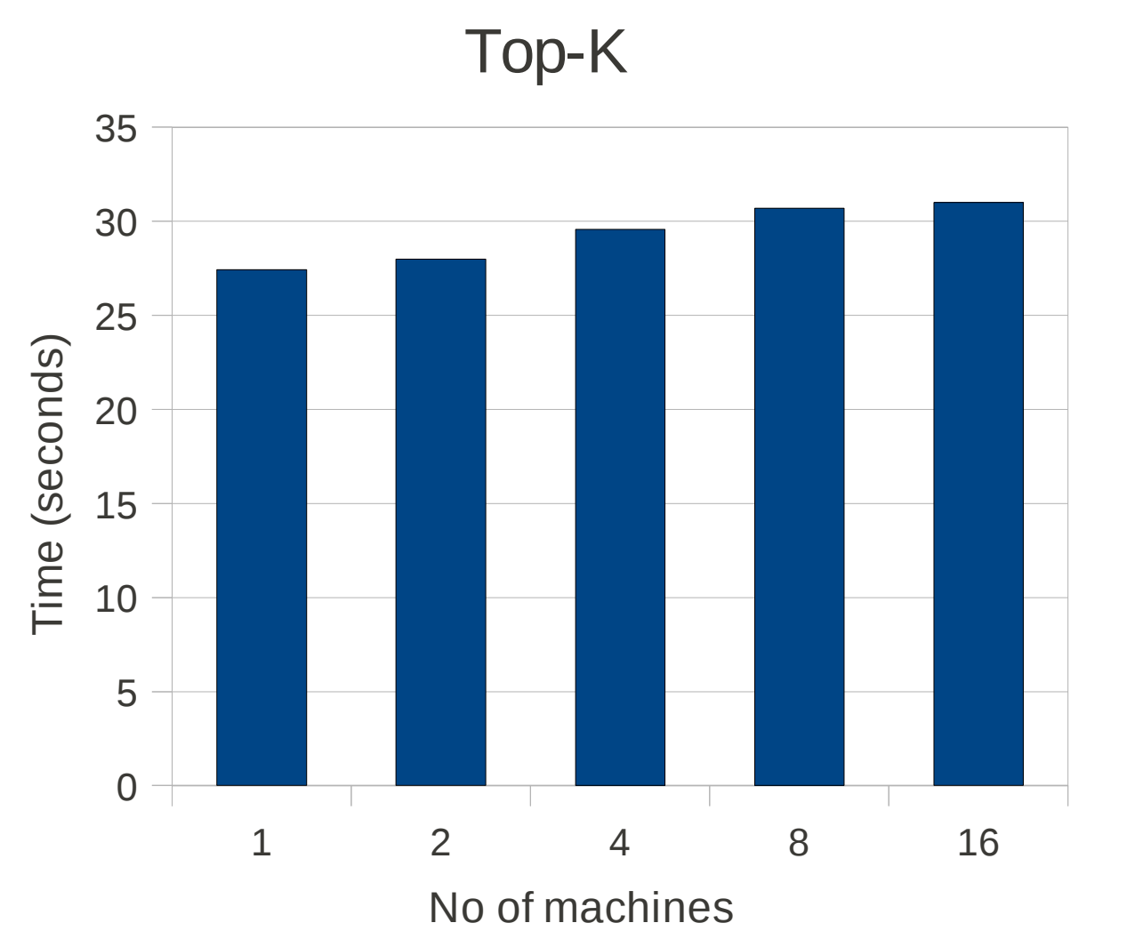
K-Means: calculates the five most representative (5 centers) ad revenues





Top-K: determines the users who generated the largest one hundred (top-100) ad revenues on a single visit


Top-K
Time (seconds)
No of disks


Top-K
Time (seconds)
No of machines