Tangram: Bridging Immutable and Mutable Abstractions for Distributed Data Analytics

Yuzhen Huang, Xiao Yan, Guanxian Jiang
Tatiana Jin, James Cheng, An Xu Zhanhau Liu, Shuo Tu
Department of Computer Science and Engineering
The Chinese University of Hong Kong
Distributed Data Analytics Systems

Distributed data analytics systems in the last decade:
- From HPC (e.g., MPI), to general-purpose computing systems (e.g., MR, Spark), to specialized systems (e.g., Pregel, Parameter Server)
Distributed Data Analytics Systems

Classification according to data abstractions

Immutable

Mutable

Parameter Server (PS)

Microsoft
DryadLINQ

Hive
Spark

Petuum
Pregel (Google)

TensorFlow
GraphLab
Immutable Abstraction

General purpose data analytics frameworks, e.g., MapReduce, DryadLINQ, Spark, etc.

• Functional programming models
• Use dataflow graphs to model the dependency among datasets

```scala
val textFile = sc.textFile("hdfs://...")
val counts = textFile.flatMap(line => line.split(" "))
  .map(word => (word, 1))
  .reduceByKey(_ + _)
```

Word Count in Spark
Immutable Abstraction

General purpose data analytics frameworks, e.g., MapReduce, DryadLINQ, Spark, etc.

+ Efficient failure recovery (lineage-based recovery)
+ Efficient load balancing (speculative execution)

- Inherently stateless
- Only support BSP (synchronous)
Mutable Abstraction

Specialized systems
• Vertex-centric graph analytics systems
  • E.g., Pregel, GraphLab, PowerGraph, etc
• Parameter-server-based machine learning systems
  • E.g., Parameter Server, Petuum, etc.

• Specialized programming models
• Stateful representation
Mutable Abstraction

Specialized systems, e.g. Pregel, Parameter Server, etc.

+ Efficient for iterative workloads
+ May support asynchronous execution
- Require a full restart from the latest checkpoint (e.g., Pregel) or use expensive replication for fault tolerance (e.g., Parameter Server)
- Rely on the nature of the applications for load balancing
Immutable and Mutable Abstractions

<table>
<thead>
<tr>
<th>Immutable</th>
<th>Mutable</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ Functional API</td>
<td>+ Stateful representation</td>
</tr>
<tr>
<td>+ Fault tolerance</td>
<td>+ Iterative and asynchronous execution</td>
</tr>
<tr>
<td>+ Load balancing</td>
<td></td>
</tr>
<tr>
<td>- Not natural for stateful</td>
<td>- Fault tolerance</td>
</tr>
<tr>
<td>representation</td>
<td>- Load Balancing</td>
</tr>
<tr>
<td>- Only support BSP</td>
<td></td>
</tr>
</tbody>
</table>

Questions
- Can we **enjoy the benefits of both worlds**?
- Can the system **transparently determine the data mutability**?
MapUpdate

MapReduce
- In the dataflow abstraction, we apply operations on collections (datasets) and generate new collections

MapUpdate
- We make data collections mutable, and change the Reduce operation to a stateful Update operation
MapUpdate

A.\texttt{map}(B, \texttt{map\_func}).\texttt{update}(C, \texttt{update\_func})
MapUpdate

A. \texttt{map}(B, \texttt{map\_func}).\texttt{update}(C, \texttt{update\_func})

Map collection \quad Side-input collection \quad Update collection
MapUpdate

A. map(B, map_func). update(C, update_func)

Map:  
- functional and immutable

Update  
- Stateful and in-place

Side-input

Map

Update
MapUpdate

Feature #1
Some or all of the map collection (A), side-input collection (B) and update collection (C) can be the same collection.

\[
A.\text{map}(B, \text{map\_func}).\text{update}(A, \text{update\_func})
\]
- map = update

\[
A.\text{map}(B, \text{map\_func}).\text{update}(B, \text{update\_func})
\]
- side-input = update
MapUpdate: Example Application

A. $\text{map}(B, \text{map\_func}).\text{update}(A, \text{update\_func})$

Vertex-centric Graph Analytics (PageRank)

```java
// Rank: (id, pr): (int, float)
// Link: (id, nb1, nb2...): (int, int, int...
map_func(Rank r, Links links, Output o):
    for each neighbor nb in links[r.id]:
        o <- (nb.id, 0.85 * r.pr/len(links[r.id]))
update_func(Rank r, float contrib):
    r.pr += contrib

ranks.map(links, map_func).
    .update(ranks, update_func).
    .setIter(30)
```
MapUpdate: Example Application

A. `map(B, map_func).update(B, update_func)`

Iterative Machine Learning (Gradient Descent)

```cpp
// Sample: (label,(k,v)..): (int, (int,float)..
// Param: (k,v): (int, float)
// data: collection<Sample>
// params: collection<Param>
map_func(Sample sample, Params params, Output o):
    grad = CalcGrad(sample, params)
    o <- grad  // grad: ((k,v)...

update_func(Param param, float update):
    param.val -= learning_rate * update
```

data.map(params, map_func)
    .update(params, update_func)
    .setIter(100).setStaleness(2)
MapUpdate

Feature #2
Supports iteration and asynchronous execution inherently

A.\texttt{map}(B, \texttt{map\_func}).\texttt{update}(C, \texttt{update\_func})
\begin{itemize}
  \item \texttt{setIter}(100)
  \item \texttt{setStaleness}(2)
\end{itemize}
MapUpdate

Feature #3
A simple mechanism to determine whether a collection is mutable in a MapUpdate plan:
• The update collection is mutable, and other collections, if different from the update collection, are considered immutable

A.map(B, map_func).update(C, update_func)
MapUpdate

Feature #3

A simple mechanism to *determine whether a collection is mutable* in a MapUpdate plan

\[
\text{A.map}(B, \text{map\_func}).\text{update}(A, \text{update\_func})
\]

- map = update

\[
\text{A.map}(B, \text{map\_func}).\text{update}(B, \text{update\_func})
\]

- side-input = update
MapUpdate: Example Application

Pipelined Workloads
- MapUpdate is especially useful for pipelined workloads
- Typical pipelines:
  - MapReduce-style data processing -> various data analytics -> testing
- Context switch overhead
Tangram

We implemented MapUpdate in Tangram
- Local Task Management
- Partition-based Progress Control
  - Support BSP, SSP and ASP execution models
  - Bitmap to record committed updates for each partition
- Context-Aware Failure Recovery

\[\text{MapUpdate} \quad \text{Tangram}\]
Context-Aware Failure Recovery

Tangram distinguishes two failure scenarios, i.e., local failure and global failure, and applies different failure recovery strategies

- Local failure: the failed machines do not hold update (mutable) partitions
  - Reloads the lost partitions (immutable) on the healthy machines in parallel and continues the execution

![Diagram](image-url)
Context-Aware Failure Recovery

Tangram distinguishes two failure scenarios, i.e., local failure and global failure, and applies different failure recovery strategies.

- Global failure: the failed machines contain partitions of the update (mutable) collection
  - Rolls back to the latest checkpoint and reloads the mutable partitions
  - Reloads the lost immutable parts in parallel
Experiments

Settings:
- 20 machines, connected with 1Gbps Ethernet
- 20 machines, connected with 10Gbps Ethernet

Experiments
- Fault tolerance for local and global failures
- Expressiveness and performance on a wide range of workloads
- Efficiency in pipelined workloads
Experiments

Failure Recovery

- Local failure: K-means
  - No need to restart from the latest checkpoint
  - Tangram took 17.8 seconds to reload the lost training data (~6GB) and finish the 7th iteration (vs. 40 seconds in Spark)
  - Similar to Spark, while other mutable systems (e.g., Naiad, Petuum, PowerGraph) have to roll back to checkpoint

(a) Local Failure
Experiments

Failure Recovery

- Global failure: PageRank
  - Roll back to the latest checkpoint (iteration 5)
  - In total, Tangram took 29 seconds to recompute the 6th iteration and finish the 7th iteration (vs. 47 seconds in Spark)
  - Spark also requires a full recomputation from the latest checkpoint in this case (i.e., long lineage with wide dependency)
Experiments

Expressiveness and Efficiency
- Bulk Processing (vs. Spark)
- Iterative Machine Learning (vs. Petuum)
- Graph Analytics (vs. PowerGraph, etc)

Results
- Tangram can express a wide variety of workloads
- Tangram achieves comparable performance as specialized systems

Word Count

K-means

PageRank
Experiments

Pipelined Workload: TF-IDF + LR

- Compared with Spark, Spark + Glint (a built-in PS), Spark + Petuum using a faster 10-Gbps network

- Spark + Petuum has high context-switch overhead
- Using Spark alone is not efficient
- Spark + Glint adds external dependencies and violates Spark's unified abstraction
Conclusions

- A novel programming model: MapUpdate
- Tangram: Enjoys the benefits of both worlds
  - Support asynchronous iterative workloads
  - Differentiated failure recovery and load balance