# Prajna: Cloud Service and Interactive Big Data Analytics

## (Distributed Platform Building Leverage Functional Programming)

Jin Li, Sanjeev Mehrotra and Weirong Zhu

Microsoft
*One Microsoft Way, Bld. 99, Redmond, WA, USA*
{jinl;sanjeevm,wezhu}@microsoft.com

## Abstract

Apache Spark has attracted broad attention in both academia and industry. When people talk about Spark, the first thing that comes to mind is the Resilient Distributed Datasets (RDDs), which lets programmers perform in-memory computations on large clusters in a fault-tolerant manner. While RDD is certainly a great contribution, an overlooked aspect of Spark lies in its harness of functional programming concept (closure, lazy computation, and delegate) in distributed system building. We believe that this is what makes Spark flexible to support different data processing scenario under one common platform and to execute the distributed operation efficiently. We also believe that Spark has just scratched the surface, and the broader use of functional computing concept can fundamentally change how distributed system is built.

In this paper, we describe Prajna, a distributed functional programming platform. Prajna is built on top of .Net and F#, and will be open sourced (pending legal review and management approval, note that all components that Prajna depends upon, including both .Net and F# have already been open sourced). Prajna not only supports (and extends) in-memory data analytics on large clusters like that of Spark, but also supports development and deployment of cloud services. Moreover, we show that Prajna can harmonize cloud service and data analytical service, and add rich data analytics on any existing cloud service/application. Prajna supports running of cloud service and interactive data analytics in both managed code and unmanaged code, and supports running of remote code with significant data components (e.g., a recognition model that is hundreds of megabytes in size). And with little programming effort (a day's work), Prajna allows a programmer to add interactive data analytics to existing cloud services and applications with an analytical turnaround time under a second. The analyzed data (e.g., statuses of the cloud service, users' inputs, etc.) is processed entirely in-memory during the analytical cycles and never stored to disk. Also, data is first locally accumulated and aggregated before further sending across the network for further cluster-wide aggregation. As such, through Prajna, cloud wide telemetry data becomes available right at users' fingertip.

***Categories and Subject Descriptors*** C.2.4 [**Distributed Systems**]: Distributed applications; D.1.3 [**Concurrent Programming**]: Distributed programming; D.4.7 [**Organization and Design**]: Distributed systems, interactive systems; D.3.2 [**Language Classifications**]: Functional languages; data-flow languages.

***Keywords*** Prajna, cloud service, interactive data analytics, distributed functional programming.

## 1. Introduction

Developed at Berkely's AMPLab, Spark [1] is a powerful data processing engine designed to handle both batch and streaming workloads in record time. On top of Spark, a number of additional modules have been built, such as Spark Stream, BlinkDB [4], GraphX [2], MLBase/MLib [3]. Existing work has shown that Spark can work with a variety of storage systems, such as Amazon S3, HDFS, and other POSIX-compliant file system, and can execute cluster computing tasks very efficiently, as demonstrated by its entry of the Gray Sort 100TB benchmark [5].

When discussing Spark, most people have focused on the Resilient Distributed Datasets (RDDs). While RDD is indeed a great contributions that enables fast distributed in-memory data processing, we feel that an overlooked aspect of Spark is its use of functional programming concept in distributed system building. While Spark has used functional programming concept such as data flow programming, lazy computation, there are other functional programming as-


*SOSP'2015* Oct 4–7, 2015, Monterey, California, USA.

pects, such as the general capability to execute a remote closure, remote delegate, that may also be very useful in distributed system building.

In this paper, we will discuss Prajna, a distributed programming platform. Prajna can be considered as a software development kit (SDK) above .Net, that makes it easier for the developer to program distributedly, be it writing cloud service to be deployed in a public cloud or a private cluster, or writing data analytical program. Prajna achieves this by leveraging the following three functional programming concepts for distributed system building.

First, Prajna enables efficient native remote execution of generic closures. In functional term, a closure is a data structure storing a functional pointer together with its execution environment and all free variables needed for the function. The contribution of Prajna is to push the boundaries of what can be included in the closures, and allow the programmer to execute any code remotely, in an environment exactly required by the code execution. Prajna accomplishes this by setting up a remote execution container, which contains the code (e.g., assemblies, DLLs, and EXEs), data, customized serializer/deserializer, execution credential, environmental variable, and remote directory that the closure expects to be executed upon. Another contribution of Prajna is to setup the container once, and repeatedly use the container as much as possible to execute remote closures. We have realized that most of the overhead of the remote execution lies in the setup of container, the launch of the container process and dynamically loading the assemblies/DLLs. Once these are done, execution of additional closures involves only operations to serialize a function pointer plus the additional data structure required for remote execution. After the closure is deserialized, each remote execution becomes a native functional call, and involves no additional interpretation.

Second, Prajna uses remote closures to launch cloud services, and allow services to import and export a contract via remote delegate. Armed with the capability to execute generic remote closures, launching a remote service in Prajna becomes the simple remote execution of a delegate `FUNC<RemoteRoleInstance>`[1] with a start parameter, where a class derived from `RemoteRoleInstance` manages the life cycle of the services. Prajna services are typically long running, and export contracts in the form of a remote delegate. In functional programming term, a delegate is a type that defines a method with strong type signature. Prajna contracts are defined as remote delegate, which appears in the form of a delegate, but can be executed locally or remotely, in one or multiple nodes. The remote delegate of contract differs from the remote closure in that no code is passed between the contract provider and the contract consumer. The code executed is always those of the contract provider. Prajna services, programs, and data analytical program can all import contracts as corresponding delegates and consume the remote function. Prajna supports task programming model, and the remote contract can be imported as `Func<Task<TResult>>` or `Func<T, Task<TResult>>`. The use of delegate wrapping to consume remote services greatly simplifies the programming between server and clients by hiding most of the implementation from the programmer. Comparing with other remoting protocols, Prajna contract has the following three contributions. 1) If the contract provider and consumer is in the same container, the execution of contract became a direct local function call, with no overhead. With this property, Prajna encourages building distributed system through componentization, by building many mini-services, as consuming contracts across services can be of zero overhead. 2) Prajna contract can be directed towards a cluster of nodes, in which one or more providers can be involved to complete the contract. This expands the remoting protocol from a peer-to-peer concept to a cluster concept, and enables a more broad categories of services [e.g., load balanced query service, Paxos service] being built on top of contract. 3) Prajna contracts can export data sequence, which can then be imported by Prajna analytical flow for analysis (see Section 3.5), and enables interactive data analytics of service telemetry.

Third, Prajna uses data flow programming concept to expose a language-integrated data analytical programming API that is similar to Spark [1]. However, while RDDs are started by transformations of data in a stable storage, Prajna's distributed data abstraction, `DSet` (Distributed data Sets) and `DKV` (Distributed Key-Value sets), can be created by running one or more closures across each machine. The resulting dataset then span into a logical collection of dataset on cluster that can be further analyzed through coarse grain transformations (e.g., `map`, `filter`, `collect`, `choose`, `mapreduce`, `sort`, `join`) and aggregation actions (e.g., `toSeq`, `fold`, `iter`). `DSet`/`DKV` greatly expands upon RDD in term of what distributed data set can be analyzed. For example, it can include: all active file storage statuses in a cluster (for distributed storage management), machine performance counter (for performance monitoring), and execution logs (for diagnosis and debugging). `DSet` and `DKV` can also be created by importing from contracts of the remote services in the form of `Func<IEnumerable<'TResult>>`. This enables Prajna to interactive analyze service telemetry without the service ever needing to persist the data. Furthermore, if the corresponding Prajna service stores the internal service data in a lock free data structure, such as those provided by `system.collections.concurrent` namespace, the data gathering thread can be executed independent of the threads that

[1] `Func<TResult>` is a .Net notation of a delegate that has no parameters, and returns a value of type `TResult`. [6]

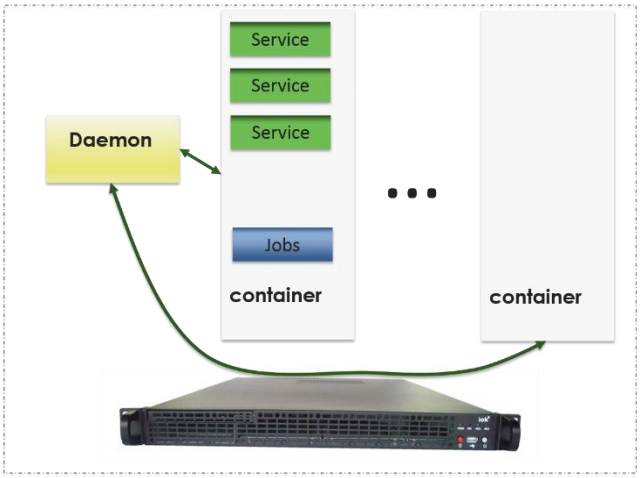


Figure 1 Prajna daemon and remote execution containers. Each container has a loopback connection links back to the daemon. If the daemon is killed, all associated containers will automatically terminate.

run the services, which do not need to slow down or even be aware that a concurrent data gathering thread is running.

The rest of the paper is organized as follows. The Prajna remote execution environment, including the daemon and the container is discussed in Section 2. We examine the services and the contracts in Section 3. The Prajna data analytical capability is covered in Section 4. We discuss distributed implementation and performance issues in Section 5. Experimental results are shown in Section 6.

## 2. Prajna Daemon, Remote Execution Container and Native Remote Execution

We discuss in this section how Prajna setups the remote execution environment, and natively executes generic closures remotely.

### 2.1 Cluster, remote execution environment and daemon.

Prajna is designed to run in the cloud, either in a public cloud, e.g. Amazon AWS, Microsoft Azure, or in a private cluster, e.g., a group of servers in a private data center. We use the term *cluster* to describe a group of machines. Most of clusters have similar characteristics, e.g., all nodes potentially locate at the same geographic regions, with similar physical machine and network properties. Nevertheless, Prajna supports multi-cluster services and cross cluster data analytics. And more than one cluster can be aggregated to form a combined cluster. The characteristics of the machines in the combined cluster may not be uniform, and there can be significant deviation in term of CPU and network performances.

The remote execution environment of Prajna is shown in Fig. 1. At each of the node that runs Prajna services and/or data analytical programs, a daemon is deployed. The responsibility of the daemon is to accommodate requests to launch services and/or data analytical jobs from clients, with the core functionality being properly setting up a remote execution container. To keep the daemon stable, it doesn't execute any customized code by itself.

### 2.2 Remote execution roster.

All remote closures are executed in the containers. Before the execution of any customized remote code, including any remote services and/or data analytical jobs, a proper remote execution *container* needs to be setup, with the following components.

***Managed Assemblies:*** Prajna automatically discovers the managed assemblies that the remote function pointer and the closure data structure depend upon. Moreover, it can automatically traverse referenced assemblies needed by the remote function for execution. Therefore, if the executed remote closure are written in managed code, there is no need for the programmer to specify the dependent assemblies, as they will be automatically discovered.

***Unmanaged DLLs, Libraries, Executables and data files:*** If the remote function calls into unmanaged DLLs, libraries, and executables, or uses data stored in data files, those files (or its containing folder) need to be specified by the programmer in the remote execution roster.

***Other dependencies:*** The programmer may specify additional dependencies that are sometimes needed in properly setting up the remote execution container. They may include: credential of remote execution, environmental variable setting, customized data serialization and deserialization delegate, and the working directory.

### 2.3 Prajna remote execution: file hash and versioning

Prajna computes a strong hash (SHA-256) for each of the managed assemblies, unmanaged DLLs, libraries, executables and data files involved in the remote execution roster. The list of hashes are sent first. Prajna daemon checks if the corresponding assemblies, unmanaged code and data files already exist in the remote node through the strong hash and looking into a common hashed directory. If the file exists, there is no need to send the file content, Prajna daemon simply links the file to its final location, and touches the original file in the hash directory. The touch operation allows Prajna to quickly track what files have been used in the remote execution environment, and garbage collect long-time unused files. If the corresponding file is not present in the remote node, the content of the file will be sent by the client. Upon receiving of the file content (be it assemblies, unmanaged code or data files), the daemon first writes the file to

**Table 1.** Interface of `RemoteRoleInstance`.

| Interface | functionality |
|---|---|
| `OnStart(param)` | Called at initialization |
| `Start()` | Called to do work, and expected to run forever |
| `OnStop()` | Called to gracefully shutdown |
| `IsRunning()` | Check if the service is still running. |

the common hash directory, and then links the file to its final location in the remote execution container.

#### 2.4 Launch of remote execution container

After the remote execution container has been setup, it will be launched as a separate process in the remote node, with proper credentials, working directory, and standard output and standard error redirected and monitored. Prajna encourages programmers to use .Net `System.Diagnostics.Trace` for most of the execution monitoring. However, there are pre-existing components (e.g., unmanaged code) that monitors execution via standard output and error, hence, Prajna also logs and monitors those.

The container establishes a fast TCP loopback [7] connection to daemon. Through this loopback connection, Prajna daemon informs the container about any client that is actively using the container to run data analytical jobs or launch services. Other services and programs can discover the container, and enumerate through the services and contracts provided by the container. Whenever this connection is severed, indicating that the daemon is closed by the user, the container will also be shutdown. The container will also be closed if there are no running services and no active clients that are using the container.

For a data analytical job that runs repeatedly (e.g., executing data analytical jobs again and again, executing machine learning tasks repeatedly), it can reuse the container that has already been setup. Prajna makes running remote closures efficient, as 1) after the container is setup, each remote closure only needs to send information of the function pointer and the data needed for the execution, and 2) if the same remote closure runs repeatedly (typical in data analytical loops or machine learning tasks), the second run of the closure becomes a direct function call.

## 3. Prajna Service and Contract

### 3.1 Services

Launching a remote service in Prajna becomes the simple remote closure execution of a delegate `FUNC<RemoteRoleInstance>` with a start parameter, where a class derived from `RemoteRoleInstance` manages the life cycle of the services. Prajna service is a long running program that provides services through either Prajna contracts (for other Prajna compatible programs, services and data analytical routines) or other standard web interface (e.g., Restful Web API [8]). `RemoteRoleInstance` exposes the interfaces shown in Table 1.

Programmers can easily convert an existing program to an Prajna service. The programming efforts involved include: extending `RemoteRoleInstance`, implementing the abstract function for initialization, doing work, shutdown, and service status, and specify the remote execution roster (see Section 2.2). Prajna service model bears resemblance to programming an Azure worker role [9]. Nevertheless, there are a number of key improvements: 1) Prajna services are launched with a start parameter, which allows the customization of services without the need to change the execution roster. 2) Prajna can recognize unchanged assemblies, DLLs, data files in the container, and avoid redeployment of those files. 3) Prajna service can be launched at any remote node a daemon is running, be it in a public cloud or a private server cluster.

### 3.2 Contracts: export/import

Prajna exports delegates as contracts, which can be imported by any Prajna compatible program, service and data analytic programs. The list of supported delegate types are shown in Table 2.

**Table 2.** Prajna contracts. (see [6] for notation)

| Type | Delegate |
|---|---|
| `Action` | No input, no return |
| `Action<T>` | Input `T`, no return |
| `Func<TResult>` | No input, return `TResult` |
| `Func<TResult>` | No input, return `TResult` |
| `Func<T,TResult>` | Input T, return `TResult` |
| `Func<Task<TResult>>` | No input, return `Task<TResult>` |
| `Func<T,Task<TResult>>` | Input T, return `Task<TResult>` |
| `Func<IEnumerable<TResult>>` | No input, return a data sequence with each element as `TResult` |
| `Func<T,IEnumerable<TResult>>` | Input T, return a data sequence with each element as `TResult` |

### 3.3 Contracts: behind the scene

Behind the scene, the implementation of Prajna contracts can be illustrated in Figure 2. When an Prajna contract provider (typically a running Prajna service) exports a contract, the contract is first installed in a local key-value store, with the contract name as the lookup key. Then, Prajna registers the contract to one or more daemons so that the contract can be discovered remotely. Furthermore, Prajna also installs a wrapped callback API for the contract so that any remote invocation of the contract by consumers clals back the delegate exported. For example, for `Func<T,TResult>`, a remote callback API is installed so that once the contract is called upon, the callback API first parses the input parameter `T` from the network stream, then calls the delegate

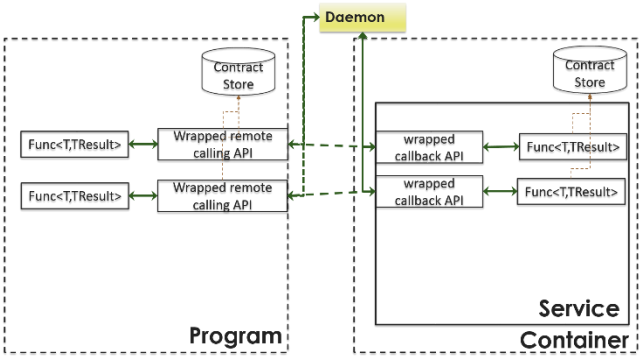


Figure 2 Prajna Contracts. The contract provider exports the contract to a local store, registers the contract, and attach proper callback API for remote execution. The contract client (can be any Prajna program, other services or data analytical routines) imports the contract by discovering it from local store and daemon. Proper wrapped delegate is installed for remote execution.

`Func<T,TResult>` that provides the service, and finally sends return `TResult` back to caller.

Prajna contracts can be consumed by any program that links to Prajna client library. At the time of import, Prajna checks if the contract with the proper name exists in its local store. If the contract is present, then the contract is checked for type signature (both input and output parameter type). If a matched contract is found, the delegate stored in the local store is simply returned to the calling application.

If the contract is not discovered in the local store, Prajna further attempts to discover the contract in the clusters (or local daemon if cluster parameter is `null`). If the contract with the proper name, input and output signatures are found, network connection is established to the contract provider, and Prajna installs a wrapped remote calling API to consume the contract service, and store the wrapped calling API to the local contract store so that the second import of the contract doesn't need to go through remote discovery again.

With this mechanism, consuming contracts among services in the same container or between data analytical jobs and services in the same container becomes a direct function call. As the returned contract is simply the same delegate exported by the contract provider. As such, Prajna encourages the componentization of services, as splitting a big services into many small component services and use contracts to consume each other leads to no overhead in final deployment.

Let us use an example to illustrate the wrapping API implementation at the contract consumer and contract provider. Say a contract of name `Contract` with calling signature `Func<T,Task<TResult>>` is being imported. After discovering the contract with remote destination `Dest`, a remote calling API of delegate `Func<T,Task<TResult>>` will be installed at the local contract store. Whenever this delegate is called with parameter `T`, the wrapping API generates a contract request and calls `Dest` with: 1) contract name `Contract`, 2) a 128-bit `GUID` [10] that uniquely identified the request, 3) input parameter `T`. The wrapping API also create a `TaskCompletionSource` [11], and immediately returned the enclosed `Task` object to the calling application. This way, the thread of the calling application does not block, and can process other works.

Receiving the contract request, the contract provider routes the request to the proper callback API based on the name `Contract`. The contract is then served, either synchronously and/or asynchronously. After the delegate successfully completes, the callback API sends back the result `TResult` together with the request `GUID`. When the returned result reaches the wrapping API of the caller, the wrapping API finds `TaskCompletionSource` of the request through the request `GUID`. After `TResult` is deserialized, `SetResult` method is called to inform the calling API that the result is available. The wrapping API also periodically checks the outstanding replies for timeouts (the timeout value can be customized for each request). Method `SetException` is used to inform the calling API that the remote delegate call fails.

### 3.4 Contracts: data serialization

For object `T`/`TResult` that is serializable, Prajna automatically uses `BinaryFormatter` to convert the object to/from byte stream. Nevertheless, if the programmer is not satisfied with either the execution performance and/or the generated byte stream size of `BinaryFormatter`, he/she can install customized serializer and deserializer to convert between object `T`/`TResult` and byte stream. Each customized serializer and deserializer is uniquely identified by a `GUID`, and the presence of the particular `GUID` alerts Prajna that the following byte stream needs to be deserialized by a certain customized deserializer.

During remote execution, the list of serializer and deserializer is sent as part of the remote execution roster to the remote node. Thus, they are available before any remote program execution.

### 3.5 Contracts for data analytics

Prajna service exports contracts of data analytical jobs in the form of `Func<IEnumerable<TResult>>` or `Func<T,IEnumerable<TResult>>`. We have given special consideration for those contracts so that data can be efficiently sent across containers, and the analytical results can be aggregated from multiple containers.

First, at the time of export, the programmer needs to set a parameter of `SerializationLimit`, which dictates how many items of `TResult` are packaged and sent together. The parameter `SerializationLimit` has no effect if the contract is being imported by a program in the same container, as that is a direct delegate call. However, if the data is being retrieved across different containers, `SerializationLimit`

governs how many items of TResult will be packaged into an array of TResult[] and sent in a single reply. The design recognizes that individual TResult object can be small, and there may be non-trivial processing overhead and serialization overhead involved. By aggregating multiple TResult items and sent them together, the overhead can be largely amortized. If the programmer desires to send each TResult one by one, he/she can set the SerializationLimit to 1. If the programmer desires to send all TResult to the calling application in one reply, he can set the SerializationLimit to a non-positive value (e.g., -1).

The customized serializer and deserializer, if used, need to be bound to the data type of array TResult[]. Because the serializer and deserializer is always working with an array of object of the same type, special compression mechanisms such as those used by SQL column store [12] or PowerDrill [13] can be employed here.

When the callback API receives a data analytical request, say of Func<T,IEnumerable<TResult>>, it calls the delegate which returns IEnumerable<TResult>. The GetEnumerator method is then called, which in most of the case starts the actual data gathering process and retrieves the twin object IEnumerator. The callback API alternatively calls method MoveNext and Current, packages and sends items of TResult[] to the calling application. When the IEnumerator reaches the end (MoveNext returns false), the remaining items (if any) are sent as TResult[]. Finally, a null object is sent. Because any valid collection of TResult items is an array of at least one, the null object uniquely signals the end of the data sequence.

Prajna allows multiple data sources from different services to be aggregated and analyzed by the same data analytical program. If there are multiple services exporting data analytical contracts of same name and signature, at the time of invocation, the calling API will issues a request to each contract provider, and records the network interface that the request is sent to. When a null object is received from a network interface, Prajna make a note that the data sequence from that network interface has completed. Only when all network interfaces have returned the null object or timed out, Prajna will finally mark the data sequence as completely retrieved, and return false for MoveNext after all returned results are iterated through.

Prajna uses lock-free data structure as much as possible to enable the calling application and the network remote functions to operate efficient yet independently without lock.

## 4. Prajna Data Analytics

### 4.1 Prajna: construction of DKV and DSet

Prajna runs data analytical jobs through a language-integrated API similar to Spark [1]. Prajna exposes DSet and DKVs, which like Spark, is read-only, partitioned collection of records across a cluster. However, unlike Spark, in which RDDs are started by transformations of data in stable storage, DSet and DKVs are started by calling one or more delegate of Func<IEnumerable<U>> on each remote node, each delegate call forms a partition on the particular remote container of DSet/DKV. DSet spans a distributed dataset, in which each element U can be considered a row in a large table. For most practical applications, the element U is a Tuple<$T_1$,$T_2$,…,$T_N$> for which $T_j$ can be considered as jth column of the row. However, there is no practical restriction of the data structure for the element of U. DKV spans a distributed key-value set, in which there is a key K and value V per row. DKV<K,V> can be interchangeably converted to DSet<Tuple<K,V>> (.Net notation) or DSet<K*V> (F# notation). DKV is mostly used for transformations that relies upon a key, e.g., distributed sort, MapReduce [20], sort join, hash join, etc..

This design enables Prajna to create distributed dataset more diversely and in-memory than what is supported by Spark. If the delegate accesses stored data in a distributed file storage system (e.g., HDFS [15], Azure), Prajna construct a DSet/DKV spanned across the distributed store. If the delegate accesses local storage of the remote node, Prajna constructs a distributed dataset through the aggregation of local storage (may be non-redundant depending on whether the local storage has been replicated) on all remote nodes. If the delegate retrieves performance counters of the running machine, Prajna forms a distributed dataset that is the aggregated performance counters of all machines of the cluster. Prajna can also import in-memory data from one or multiple services that runs in the same or different container of the data analytical job to from DSet/DKV, as shown in Section 3.5.

### 4.2 DSet/DKV: partitions and collections.

In Prajna, each DSet/DKVs consists of a set of partitions, each partition is resultant from the running of a delegate on one particular remote container. Each DSet/DKVs has a partition mapping, which records mapping information between partition A and remote node B.

For example, if the partition is resultant from iterating through a non-replicated local storage, through performance counters, and/or through importing contracts of Prajna services running on the same remote machine, as discussed in Section 3.5, the partition is uniquely mapped to the particular remote node that the delegate runs. The failing of the remote node renders that particular partition unavailable. On the other hand, if the DSet/DKVs is resultant from accessing a remote distributed redundant store, such as HDFS/Azure, each partition can be accessed by any remote node in the cluster. If the DSet/DKVs is resultant from Prajna natively implemented triple replicated store, the partition is accessible on the three nodes that the replicated stream is stored at.

Based on the particular partition mapping structure, DSet/DKV may or may not be resilient. If DSet/DKV is formed

from a reliable distributed storage (e.g., triple replicated local store, HDFS and/or Azure), the resultant DSet/DKV is resilient, because the failing of some nodes in the cluster will not affect the reliable retrieval of the data. However, if DSet/DKV is spanned in other mechanism, e.g., via delegate on non-replicated local storage, on performance counters, on contracts of Prajna services running on the same remote machine, the resultant DSet/DKV is not resilient, as failing of nodes will render the associated local storage, performance counters, and/or Prajna services information on those node inaccessible. For this reason, we call the dataset on Prajna as DSet (Distributed data Sets) and DKV (Distributed Key-Value sets). The behavior of inaccessible data of the failing node is fine and in most cases desirable, as the machine statuses of the failing node need not be analyzed in most of the cases.

A DSet/DKVs partition is further split into collections, each collection holds an array of items of U[], with the size of the collection governed by parameter SerializationLimit. Like we have explained in Section 3.5, the collection concept allows multiple small items to be aggregated into an array, thus amortizes processing overhead (e.g., type cast, check for cancellation, etc.) and improves serialization overhead. The parameter SerializationLimit can have a significant impact on performance, as too small value may lead the high overhead as each data item is processed through different stages, and a too big value may lead to poor CPU cache (L1/L2/L3) hit rate as the collection cannot fit in the cache.

### 4.3 DSet/DKV: per collection transformations.

Prajna can further create DSet/DKV from other DSet/DKVs. These operations are called *transformations*. The transformed DSet/DKVs are not materialized immediately. Instead, they are lazily evaluated in the cluster. Prajna records information on how the transformed DSet/DKVs are constructed through other DSet/DKVs, and the content of DSet/DKVs are only materialized during execution, and are immediately released if it is not further used. Prajna triggers execution of data analytics through actions, such as ToSeq() and fold. Due to space limitation, we will only examine a selected list of Prajna transformations and actions through the rest of the section, and comment on the characteristics and implementation of some of them. Please refer to Prajna API document for more complete information of Prajna transformations and actions offered.

### 4.4 DSet/DKV: per collection transformations.

A selected set of Prajna per-collection transformations is shown in Table 3. These transformations share common behaviors in that the transformed DSet/DKVs have the same partition and collection structure as the source DSet/DKVs.

We design the signature of the per-collection transformations to be similar to the corresponding modules in F# Collections.Seq module. Thus, Map creates a new DSet/DKV whose elements are the result of applying a given function to each of the element of the prior DSet/DKV. Filter returns a new DSet/DKV containing only the elements for which the predicate delegate returns true. Choose applies the given function to each element, and returns a DSet/DKV for each element where the function returns Some (of F# Options) with value. Collect applies the delegate to each element, and concatenates result to a new DSet/DKV.

Prajna supports both F# style asynchronous workflow through AsyncMap and .Net style Task Parallel Library (TPL) via ParallelMap. In either of the cases, a threadpool will be used parallel execute work items asynchronously.

**Table 3.** Prajna per-collection transformations

| Type | Delegate | F# notation |
|---|---|---|
| Map | Func<U,V> | U→V |
| AsyncMap | Func<U,async<V>> | U→async<V> |
| ParallelMap | Func<U,Task<V>> | U→Task<V> |
| Filter | Func<U,bool> | U→bool |
| MapByCollection | Func<U[],V[]> | U[]→V[] |
| Choose | Func<U,V option> | U→V option |
| Collect | Func<U,IEnumerable<V>> | U→seq<V> |

### 4.5 DSet/DKV: in-node transformations

A selected set of in-node transformations is shown in Table. 4. The transformations in this category change the collection structure of the input DSet/DKVs, but do not alter the partition structure. RowsReorg restructure the collection with a new SerializationLimit parameter. In particular, RowsSplit splits the collection to contain a single item (i.e., U[] with an array size of 1). RowsMergeAll merges all collections of the partition into a single partition, which can be used for per-partition operation such as the operation required for the Reduce step in MapReduce [20] or distributed sort. Split applies delegate to data item U, and split a DSet/DKV into multiple DSet/DKV, each of which can be considered as a column in the original dataset. Merge reverses the operation of split, and merge multiple DSet/DKV into a single DSet/DKV with a per-row merging delegate. Prajna also supports a number of sorted join (also called merge-join) operation, including inner sort join, left outer sort join and right outer sort join [21]. In all the sort join operation, we assume that two operand DKVs have already been distributedly sorted, across and within partitions, and they both use the same partition delegate so that an element of a certain key in either of DKVs always exist in one particular partition.

**Table 4.** Prajna per-collection transformations

| Type | Function |
|---|---|
| RowsReorg | Change collection size |
| RowsSplit | Split collection to single element |
| RowsMergeAll | Merge elements in partition to a single collection |
| Split | Split a DSet/DKV to multiple |
| MixBy | Merge split DSet/DKV |
| InnerSortedJoin | Inner sorted join |
| RightOuterSortedJoin | Right outer sorted join |
| LeftOuterSortedJoin | Left outer sorted join |

#### 4.6 DSet/DKV: cache transformations

A selected set of Prajna cache transformations is shown in Table. 5. The CacheInRAM transformation hints that the created DSet/DKVs should be kept in memory after it has been computed, while CacheInRAMDictionary also creates an additional ConcurrentDictionary data structure to hold data. The later becomes useful to speed up the sort join operation, if the joined DKVs can be completely cached in memory.

Table 5. Prajna cache transformations

| Type | Function |
|---|---|
| CacheInRAM | Cache DSet/DKV for repeated data analytical operation |
| CacheInRAMDictionary | Cache DKV for repeated data analytical operation, in particular fast hash join operation |

#### 4.7 DSet/DKV: cross-node transformations

A selected set of Prajna cross node transformations is shown in Table. 6. These transformations generate DSet/DKVs with different partition structure, often sends data across nodes in the cluster. Using a delegate, Repartition redistributes each data element to a new partition. MapReduce implements well-known MapReduce transformations in [20]. Sort implements a distributed bin sort, in which a partition delegate determine which partition the key should be placed, and then sorted DKV within the partition using a comparer delegate. HashJoin combines two DKV based on a key with equality delegate. CrossJoin returns the Cartesian product of rows from the two DSet/DKV.

Table 6. Prajna cross-node transformations

| Type | Function |
|---|---|
| Repartition | Repartition a DSet/DKV |
| MapReduce | MapReduce (see [20]) |
| Sort | Distributed Sort |
| HashJoin | Hash Join |
| CrossJoin | Cross join |

#### 4.8 DSet/DKV: Actions

All above DSet/DKVs operations are lazy transformations, where the resultant DSet/DKVs are generated on demand. That means that the programmer should not attempt to time any of the above transformations, as the resultant time only show the time to construct the execution graph in Prajna. Only actions evaluate and manifest DSet/DKVs, and a selected set of Prajna actions is shown in Table. 7. The action SaveAndMonitor evaluates DSet/DKV, write to a distributed file system, and monitor the progress of the operation. Internally, SaveAndMonitor first splits the DSet/DKV, and directs one DSet/DKV to a ToSeq action to be monitored, and directs the other DSet/DKV to a Save transformation to be written to a distributed file system. Prajna performs save operation this way to make it possible to write to multiple DSet/DKVs in a single action. This can enable a database columnar store like save operation, where a large table of DSet/DKV is split into many DSet/DKV of a single column, each of which is stored in a separate stream, enable more efficient compression [12][13] of columnar byte stream. Also, subsequent operation can selectively load columns of DSet/DKV, using MixBy transformation to generate a new DSet/DKV a chosen set of columns. ToSeq operation manifest DSet/DKVs into an IEnumerable<U>. ToSeq operation itself is a transformation, as the DSet/DKV hasn't been evaluated in that point (so don't time that operation). However, subsequent action on IEnumerable<U>, in particular the GetEnumerator method will trigger the evaluation of DSet/DKV. Fold and Iter are two distributed aggregation actions on DSet/DKV. Fold evaluates DSet/DKV, applies an accumulator function that accumulate the result per partition, and then applies an aggregation function to aggregate results across all partitions. During execution, each Prajna node will first execute accumulator function per partition, and then aggregate all results within that node. The final result is send back to the calling node, which aggregates result across nodes. Iter applies a delegate to first transform the element to an aggregable data structure, the result is then aggregated within and across partitions. Internally, Iter is implemented via Fold.

Table 7. Prajna actions

| Type | Function |
|---|---|
| SaveAndMonitor | Write to a distributed file system, and monitor the progress of the operation |
| ToSeq | Manifest the result as an IEnumerable<U> |
| Fold | Apply a accumulator functional per partition, then further aggregate the results of all partitions |
| Iter | Apply a delegate to each element, and aggregate the result within and across partitions |

### 4.9 DSet/DKV: behind the scene

In implementation, all DSet/DKVs transformations only setup an execution graph in the calling application. There is no network call to other node, and none of the DSet/DKVs involved are being evaluated. The only operations that trigger a distributed Prajna data analytical computation are the actions. (for ToSeq, that is the GetEnumerator method call).

When the calling application encounters an Prajna action, an execution graph is materialized, with all DSet/DKVs involved in the computation. The delegate within each of the transformation and action is also captured as a closure, with function pointer plus the additional data structure required for remote execution. They are serialized together with the execution graph. The execution graph is sent to each of the remote container that will execute the data analytical program. Upon receiving the execution graph, the remote container instantiate DSet/DKVs, to reconstruct the execution graph in each of the container of the remote node. The one exception is for any DSet/DKVs that are involved in cache transformations. Prajna remote container will attempt to look for the cached DSet/DKVs objects in the remote container, and if found, reuse the cached DSet/DKVs (and their data) for the following computation.

During execution, the DSet/DKVs that are being read (by ToSeq/SaveAndMonitor) or being accumulated by (Fold/Iter) spins up M repeatable work items, each of which is cued to one of N threads for possible execution. The parameter N can be controlled by NumParallelExecution parameter. For each work item, a collection worth of items is read or accumulated upon, and if the DSet/DKVs is a transformation, the upstream DSet/DKVs is called, with the read/accumulation operation wrapped as a continuation operation. If the upstream DSet/DKVs is still a transformation, the transformation operation itself is wrapped as a continuation (with the read/accumulation operation further wrapped). Please note that wrapping as continuation does not produce additional stack, especially as F# optimizes tail recursive calls. When the operation reaches the final source DSet/DKVs that are reading from distributed file, or importing data from contracts from other Prajna services, the continuation operations are unwrapped and executed upon. This style of program execution is similar to what has been implemented in in F# Collections.Seq module. For Prajna per-collection transformations as shown in Table. 3, a collection worth of data is read/imported, operated and passed by each transformation, and finally read (by ToSeq/SaveAndMonitor) or being accumulated by (Fold/Iter). The entire DSet/DKV, and/or even the entire partition of DSet/DKV is never instantiated in the memory. This programming model enables Prajna to process with huge distributed data set efficiently.

### 4.10 DSet/DKV: distributed failure recovery

A programming semantics that Prajna shares with that of Spark is that DSet/DKV are immutable in the data analytical operations. Thus, when a node fails during the computation, only the failed partitions of DSet/DKV need to be identified and recomputed, without the need to roll back the entire computation.

Refer to Section 4.2, depending on how the delegate runs to create the DSet/DKV, the source DSet/DKV has different behavior in its capability to deal with failed partition. If the partition is uniquely mapped to a particular remote node, such as resultant from iterating through non-replicated local storage, through performance counters, and/or through contracts of Prajna services running on the same remote machine, the failure of the remote node will leads to that particular partition to be removed from the computation. If the partition is created from accessing a remote distributed redundant store, such as HDFS/Azure, the partition can be recreated by any remote node other than the failed node in the cluster. If the partition is accessed through the triple replicated local store, the failed partition can be accessed from the other node that is not failing. After the partition of the failed node have been remapped, Prajna re-compute the failed partition using a mechanism similar to RDDs in Spark[1].

## 5. Prajna: Debuggability, Performance, and Code Rewrite

Working on a large distributed system such as Prajna brings a number of challenges in system implementation. We choose to discuss three issues in this section. They are: 1) system debuggability, 2) performance related to task parallelism, and 3) the need to rewriting a significant portion of code.

### 5.1 Debuggability

First and foremost, we have found that the debuggability becomes a core competency in the system development. When dealing with a super complicated distributed system such as Prajna, bugs became unavoidable and may bury deep in the system. We have placed catch strategically in code so that we can localize exception. Moreover, we have found the following two tricks to be particularly helpful.

#### 5.1.1 Delegate as trace Func<string>

We have placed trace log liberally throughout the code to monitor the operation flow and execution path of code. However, we want to make sure that those trace operation do not slow down the code execution if we are sufficiently confident that the particular code segment is not the one that causes problem. For that purpose, we program each trace operation as a delegate guarded by a certain trace level, as:

```
Trace (level, Func<string>) (.Net notation)
```

`Trace level ()→string (F# notation)`

In either of the forms, if the guarded level is above the current trace level (a global static variable), the delegate which evaluate the trace output will not be executed. And the operation overhead is simply a comparison again a global static variable (an in-cache comparison operation). Therefore, we can simply lower global trace level, or raise the trace level of a particular code groups to turn on/off traces in code dynamically, while incur minimal overhead when the trace is not outputted (as most of the heavy computation is in evaluating trace output).

#### 5.1.2 Dynamically attach IDE for debug

We build Prajna deployment environment in such a way that the `PDB` of Prajna daemon and container are also deployed to the remote node. We run Prajna daemon and container in `Debug` mode, as we observe that .Net incurs minimal overhead running code in `Debug` mode vs running code in `Release` mode. When we find issue, we develop capability of Prajna to attach a debug IDE (Integrated development environment) to monitor the inner operation of Prajna daemon and container. We have found that this approach significantly speed up distributed debugging, as we have the full functionality of IDE (including setting breakpoint, watch, etc.) at our disposal.

### 5.2 `Performance related to task parallelism`

Prajna is built as a high performance distributed system. The core engine of Prajna uses lock-free data structure in `System.Collections.Concurrent` space, and adopt a task parallel execution strategy. In the beginning, we have attempted to use `F#` async workflow or task parallel library (TPL) to implement the Prajna execution engine. However, we have observed severe performance issue in the implementation. In some execution, we observe async DNS resolve takes 70 seconds to complete, which causes a ripple effect in the system performance.

We localize the issue to the architecture of async workflow and task parallel library. Both are designed and require all work items to be implemented asynchronously. However, in actual implementation, this may be difficult to accomplish. For example, there are asynchronous write, read, and flush method related to `FileStream`, however, there is no native asynchronous method to create a directory, change file attribute, etc.. When some (even small number) of blocking operations are mixed with asynchronous workflow, it could lead to a significant performance degradation of the entire system, as threads are taken out of execution pool by the blocked task or work item. This performance issue is particularly annoying as the performance hit happen sporadically. For example, say among a set of tasks scheduled, a small set of task `A` blocks, and another set of task `B` can unblock those task `A`. In many execution cases, at least some of the tasks `B` are executed before `A`, and the performance degradation is not so bad. However, in one particular run, the task scheduler may happen to schedule all task set `A` before task set `B`, and suddenly, a severe performance degradation is observed. .Net is supposed to observe the performance degradation and launch additional thread to compensate, but we observe that in many common scenarios, .Net doesn't launch thread fast enough.

After pinpointing the issue, we have re-implemented the execution engine with a customized threadpool with work items that is described in Section 4.9. The work item is like `Task`, but we have used two additional mechanism to diagnostic blocking and performance degradation issue. First, we require each work item to provide a delegate that provides debugging information (in the form of `Func<string>` or `()→string`). When Prajna observes that the work item has been executing a long time in the execution engine, the debugging information will be printed in trace to identify such work items with performance issue (and potential blocking information). Second, we provide mechanism to let programmer to signal that the code is executing potential blocking code via `EnterBlock` and `LeaveBlock` pair, or is waiting for a `WaitHandle` to fire via `SafeWaitOne` call. Internally, Prajna monitors if the blocking code is run among one of the customized thread pools, and if a large number of threads are blocked. If both is true, additional threads will be launched to compensate for the blocked work items, and reassume the execution.

### 5.3 F#, distributed system development and code rewrite

At this stage, the code base of Prajna consists of 50,000 line of code (LOC), including significant comments and trace and debugging code. The core development team is two people. Prajna is developed based on .Net 4.0 and F#, but there are no other code dependency.

The extreme compact code base speaks volume of compactness nature of F# code. Especially, we find that being strong typed with type-inference capability allows the developer to focus energy on the core logic of the code. We have need to rewrite the core execution engine of Prajna a couple of time (e.g., a significant rewrite is to use the customized threadpool as execution engine.) With F#'s help on the strong typed and type-inference, we can complete the rewrite of the core execution engine without touch the rest of the code base. During many of the small rewrite (in term of the code regions touched), we observe that once the code compiled through, it just works and passes the dozen of unit tests we have designed (including distributed write, read, failure recovery test, MapReduce, distributed sort, sorted join, hash join, cross join, etc..)

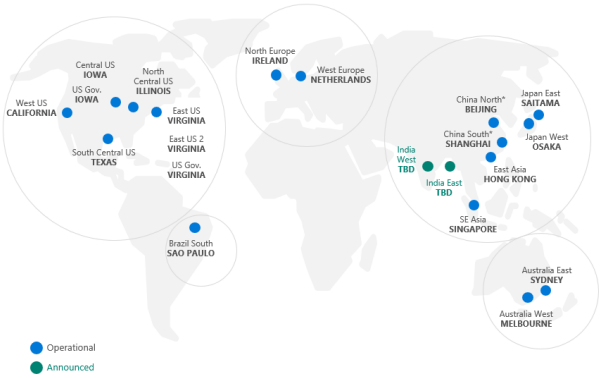


Figure 3 VM Hub Front End Deployment location.

## 6. Experimental Results

In this section, we show how Prajna is used to build cloud service. We also show how to add interactive data monitoring capability on the Prajna services running in the cloud.

### 6.1 VM Hub, a cloud based visual recognition hub

We worked with our colleagues to develop Visual Media Hub (VM Hub), which is a generic visual recognition hub deployed in cloud. VM Hub consists a front end service and a back end service. The front end service receives recognition requests of image/video frames through a Restful WebAPI [26]. The front end service also includes a Web interface for human operator to monitor the health of Front end operations. It then sends the image/video frame to one of the back end for recognition. The back end service runs an object recognition instance, and export an image/video object recognition contract to a set of front ends.

Writing and deploying services using Prajna is easy. Programmer simply wrote the service using his/her favorite platform/tools, and then wraps it up with Prajna. The wrapping includes spelling out all elements of the Remote Execution Container, and then write a delegate and start parameter that launches the remote services (see Section. 2). For the front end, as the services is completely in managed code, Prajna automatically traverses and discovers all assemblies required for the service. The programmer does need to spell out a data directory needed in the launch of front end service. We further use a traffic manager to load balance across the cluster of front end, and direct user recognition request to the closest front end node.

The back end recognition service is designed by our colleagues. It has capability to recognize 11 categories of image objects: dog breed, Orlando/Austin/Seattle/Beijing /Sichuan landmarks (one category for each landmark), Disney attractions, bottled drinks, photo tagging, flowers, and office products. The recognize is mostly unmanaged C++ code, and include components such as Caffe recognition engine [27], Cuda [28], Open CV library [29] and LevelDB [30]. Moreover, each recognition category includes a model directory. All together, the remote execution containers of the recognition service includes 40 files in addition to the assemblies. They include 7 files in the model directory (in particular the Caffe recognition model is usually around 200MB, and takes bulk of the size of the remote execution container), 21 DLLs, 3 exes. For proper execution, the code also expects a particular directory structure at remote for the model file. Also, if the standard output and standard error of the remote container is not properly hooked up, the code will not run.

We write a thin wrapper (~400 line of code) to hook this recognition service to a VM Hub back end recognition service. Once wrapped, the back end recognition service home in to the cluster of VM Hub front end server to provide image recognition service. The Caffe recognizer is a single instance recognition engine (e.g., the recognizer only process one image at a time). Prajna though may launch multiple instances on a single machine, so that each back end server can serve multiple image recognition request. We do observe that the inner code of Caffe recognizer contains parallelized CPU execution that use all cores on the target machine, and use Cuda if GPU is available. Thus, when multiple recognition instances are running (each in its own remote execution container), every recognition instance slows down significantly.

We have launched a set of VM Hub front ends running in each of the deployed location of a major public cloud provider (blue dots in Figure. 3). We have then run 80 back end instances, where 60 instances are on an old cluster and 20 instances are on a new cluster, see Table. 8. Any one of the front end server or back end server can fail independently at any moment. The Prajna supported VM Hub will be able to provide continuous image/video recognition service as long as at least one of the front end server and back end server is live. With this experiment, we show that Prajna is capable of deploying complicated service writing by others (includes both managed and unmanaged code), and handle remote deployment of service in both public cloud and private cluster.

Table 8. VM Hub: Front End / Back End configuration

| Type | Specification |
|---|---|
| `Front End` | VM, 2 cores, 3.5GB |
| `Back End 1` | Operon® 2352, 2100 Mhz, 4 Cores, 8GB RAM |
| `Back End 2` | Dual Proc Xeon® E5-2450L, 8 cores per processor, 192GB RAM |

### 6.2 Real-time interactive data analytics

The back end and front end services export contract of information to be analyzed. The list of contracts exported by the front end services and back end services are shown in Table. 9.

Table 9. VM Hub: Exported contracts

| Exported by | Contract |
|---|---|
| Front end | Network RTT (to back end) |
| Front end | Expected service latency by back end |
| Front end | Back end service queue (capacity, current load, # of request from this front end) |
| Front end | User request (including image/video recognition request and all user activities on the Front End web) |
| Front end | Request statistics (time received, recognition category, web activity tag, recognition performance break down [assignment, network, queue, processing] ) |
| `Back end` | Network RTT (to front end) |
| `Back end` | Recognition category |
| `Back end` | Request statistics (front end servers that send in the request, recognition category, service status, recognition performance breakdown [queue, processing]) |

A typical block of codes that perform the data analytics on the front end or back end services are as follows. We use the calculation of aggregation recognition statistics, e.g., 99.9-percentile recognition latency as examples. In the front end / back end service, we have code:

```
exportSeqFunction ("perf", getStatistics)
```

Internally, the front end / back end service uses a lock-free `ConcurrentQueue` to hold a recent statistics of 10 minutes worth of recognition queries. The object recognition thread can continuously add query to the statistics store, dequeue those statistics that are more than 10 minutes old. The method getStatistics simply casts the internal query statistics store to `IEnumerable<_>`. When the data analytic program is running, it will call back and gets the `IEnumerable<_>` object of the statistics store, and enumerate through the statistics. Because the statistics store is lock-free, none of the thread needs to get an exclusive lock to access the query statistic store (either adding statistics, removing timeout statistics, or enumerating through statistic stores).

The data analytical program has codes:

```
let a = DSet<_>.import cl null "perf"
let b = a.RowsReorg -1
let c = b.MapByCollection queryPerformance
let d = c.Fold perfAgg perfAgg null
```

The first line of code imports the contract from the front end and/or backend service. The first parameter is the back end/front end cluster that the analytics will be performed upon. The second parameter indicates that we are importing contracts from the daemon, which automatically imports all contracts of services that linked to the daemon (i.e., all instances of the front end / back end that runs on that machine.) The third parameter is the name of the contract. At this moment, we have created a `DSet` that span across the entire front end or back end clusters, with one partition on each of the node. The second and third line of code aggregate all data in a partition to a single collection, and then computes an aggregated query performance. Because we are calculating aggregation statistics that needs to sort through the collection of recognition query to obtain medium, 90-percentile and 99.9-percentile performance, we first aggregate all query statistics to a single collection, and then maps this collection to one single aggregated statistics. The forth line of code first accumulates the statistics in each partition (which does nothing as there is only one aggregated statistics), and then aggregates the statistics across back end / front end nodes. All computation until accumulating the statistics in each partition is done locally in the back end / front end nodes where the query statistics is located. Only at that moment, the aggregated statistics per node is serialized and sent through the network to be aggregated by the monitoring node. Prajna is capable of providing low latency, high performance data analytics because it performs most of the big data computation in memory (this even includes the collection of the source data to be analyzed). The data never touches the disk during the entire cycle of computation.

## 7. Related Works

In this aspect, Prajna is similar to many other remoting protocol (e.g., Java's RMI/JRMP, .Net remoting, Windows Communication Foundation). Nevertheless, by using delegate, Prajna contract has three key features that differentiate it from other remoting protocol.

The programming model of Prajna is heavily influenced by Spark [1]. The core engines of Prajna and Spark are both written in a functional programming language (Scale in Spark, and F# in Prajna). Some of the programming concept of Prajna, e.g., executing remote code via closure, `DSet/DKVs` being immutable, lazy instantiation of transformed distributed data set, failure recovery by only re-computing the failed partition, is influenced by Spark.

However, Prajna goes steps further in leveraging functional concept in distributed system building. It realizes that the native remote execution mode of closure execution can be not only used in data analytics, but also can be used in launching applications and staging remote services. It further expands the native remote execution to include not only managed code, but also unmanaged code (with DLLs) and data files. It allows `DSet/DKVs` to be created by running customizable code on remote machine, and by importing from contracts in other machines. Though the data analytical pro-

gram of Prajna still uses immutable `DSet/DKVs` in its programming flow, mutable data set (such as an in-memory key-value store) can be hosted in one of the Prajna services, thus significantly expand the programming modality that can be supported by Prajna.

While Spark significantly speeds up the building of distributed data analytical program, Prajna has the potential to speed up the building of distributed data analytical programs and distributed services, and combine both services and data analytical program under a single umbrella. For example, there are other high-level distributed programming interfaces that are available, e.g., distributed Actor framework in Orleans [23], geo-distributed database in Spanner [24] and distributed machine learning [25]. Each offers interesting cluster capability, but each is usually implemented in a different programming paradigm and implementation framework. With Prajna's core providing a flexible native remote execution model, a remote closure execution and contract provider-consumer interface, and a cluster data analytical feature, it becomes possible to quickly build those distributed Actor, geo-distributed database, and distributed machine learning capability under Prajna. It may significantly reduce the engineering time needed to build a distributed service.

## 8. Summary

In this paper, we describe Prajna, a distributed functional programming platform that can be used to quickly develop both services (deployable in public and in private cloud) and interactive data analytical capability of the running services. Extensively using functional programming concept, Prajna significantly improve the programmability, the debuggability and the performance of distributed system building. Prajna adopts a native remote execution model for running remote closures (functional pointer plus additional data structure). The assemblies, DLLs, and data files needed for remote execution is sent to launch a remote container, and only updated whenever one of the file content changes (verified by a strong hash). Prajna launches services through a delegate with parameter, and allows service to export a variety of contracts that can be consumed by other programs, services and/or Prajna data analytical programs. Prajna can execute big data query across a wide cluster interactively across a large cluster.


[1]. Zaharia, Matei, et al. "Spark: cluster computing with working sets." Proceedings of the 2nd USENIX conference on Hot topics in cloud computing. 2010.

[2]. Gonzalez, Joseph E., et al. "Graphx: Graph processing in a distributed dataflow framework." *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2014.

[3]. Talwalkar, A., et al. "Mlbase: A distributed machine learning wrapper." *NIPS Big Learning Workshop*. 2012.

[4]. Agarwal, Sameer, et al. "BlinkDB: queries with bounded errors and bounded response times on very large data." *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM, 2013.

[5]. Xin, Reynold, et al. "GraySort on Apache Spark by Databricks.", http://sortbenchmark.org/ApacheSpark2014.pdf.

[6]. Petricek, Tomas, and Jon Skeet. *Real World Functional Programming: With Examples in F# and C#*. Manning Publications Co., 2009.

[7]. Ed Briggs. "Fast TCP Loopback Performance and Low Latency with Windows Server 2012 TCP Loopback Fast Path.", http://blogs.technet.com/b/wincat/archive/2012/12/05/fast-tcp-loopback-performance-and-low-latency-with-windows-server-2012-tcp-loopback-fast-path.aspx

[8]. Richardson, Leonard, and Sam Ruby. *RESTful web services*. " O'Reilly Media, Inc.", 2008.

[9]. Brunetti, Roberto. *Windows Azure step by step*. Microsoft Press, 2011.

[10]. http://en.wikipedia.org/wiki/Globally_unique_identifier

[11]. Blewett, Richard, and Andrew Clymer. "Everything a Task." *Pro Asynchronous Programming with. NET*. Apress, 2013. 149-160.

[12]. Larson, Per-Åke, et al. "SQL server column store indexes." Proceedings of the 2011 ACM SIGMOD International Conference on Management of data. ACM, 2011.

[13]. Hall, Alexander, et al. "Processing a trillion cells per mouse click." *Proceedings of the VLDB Endowment* 5.11 (2012): 1436-1446.

[14]. Danny Shih, Parallel programming with .Net, http://blogs.msdn.com/b/pfxteam/archive/2010/01/26/9953725.aspx

[15]. Borthakur, Dhruba. "HDFS architecture guide." *Hadoop Apache Project* (2008): 53.

[16]. Calder, Brad, et al. "Windows Azure Storage: a highly available cloud storage service with strong consistency." *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 2011.

[17]. Syme, Don, et al. "The F# 3.0 Language Specification." (2005).

[18]. Petricek, Tomas, and Don Syme. "Syntax Matters: Writing abstract computations in F#." *Pre-proceedings of TFP (Trends in Functional Programming), St. Andrews, Scotland* (2012).

[19]. Leijen, Daan, Wolfram Schulte, and Sebastian Burckhardt. "The design of a task parallel library." *Acm Sigplan Notices*. Vol. 44. No. 10. ACM, 2009

[20]. Dean, Jeffrey, and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters." *Communications of the ACM* 51.1 (2008): 107-113.

[21]. http://en.wikipedia.org/wiki/Join_(SQL)

[22]. https://msdn.microsoft.com/en-us/library/system.tuple(v=vs.110).aspx

[23]. Bernstein, P., et al. *Orleans: Distributed Virtual Actors for Programmability and Scalability*. MSR Technical Report (MSR-TR-2014-41, 24).

[24]. Corbett, James C., et al. "Spanner: Google's globally distributed database." *ACM Transactions on Computer Systems (TOCS)* 31.3 (2013): 8.

[25]. Dean, Jeffrey, et al. "Large scale distributed deep networks." *Advances in Neural Information Processing Systems*. 2012.

[26]. Richardson, Leonard, and Sam Ruby. *RESTful web services*. "O'Reilly Media, Inc.", 2008.

[27]. Jia, Yangqing, et al. "Caffe: Convolutional architecture for fast feature embedding." *Proceedings of the ACM International Conference on Multimedia*. ACM, 2014.

[28]. Nvidia, C. U. D. A. "Programming guide." (2008).

[29]. Bradski, Gary, and Adrian Kaehler. *Learning OpenCV: Computer vision with the OpenCV library*. " O'Reilly Media, Inc.", 2008.

[30]. http://en.wikipedia.org/wiki/LevelDB