Distributed and Microservices support to ScalaTion

Vinay Kumar Bingi (vinaykuma.bingi25@uga.edu)

December 15, 2018

1 Introduction

Though being around from long time, distributed database systems [9] are gaining popularity in recent times with the increase in size of data that the modern day applications are using. These systems are evolving dramatically with growing in use of unstructured data and NoSQL DBMS [10]. Not just the ability to accompany the increase in size in the data, but also the ability to use heavy computation systems, reducing the load of tasks on individual systems, a possibility of multiple replications of data keeping the data protected and multiple other reasons adds to increase in the use of distributed database systems. Even though they come with complexities and difficulties to work with, they still are reliable in processing and storage of databases.

Microservices architecture [2], on the other hand, is again a recently popularized term in software architecture. Inspired from service-oriented computing, microservices architecture includes splitting of the application into smaller cohesive and independent services that can run as individual processes [3]. Each of these individual process is called a microservice. The advantages being several, the microservices architecture is used to get rid of a monolithic system. Microservice architecture can work as a distributed application making it easier to scale up the applications. Each service can be set up or can be requested only when needed and avoid to load all the modules of the system into memory.

This work is related to providing a distributed databases support and microservices architecture on different modules of ScalaTion [7].

2 ScalaTion

ScalaTion is a Big Data framework which provides support for analytics, simulation and optimization. The software includes modules on mathematics, statistics, databases, and modeling. It also provides several real-world analytics and simulation models as examples.

The `scalation_database` module provides support to columnar database alongside graph and spatial databases. The columnar databases are built on `Vector [Vec]`. Where `Vec` is a collection of vectors of multiple data-types - *Int, Long, Double, Real, Complex, Rational, String, Time*. Each vector is a column of the database relation. The module provides support for relational algebra and other processing and analytic operations on the relations.

The `scalation_modeling` module has a huge collection of analytics and simulation algorithms. It includes linear models, regression models, neural networks, classifiers, clusterers, forecasting models and others.

The current version of ScalaTion is a monolithic in-memory software. This brings its limitations on scaling in certain applications leading to need of heavy processing and memory and tackling time constraints. This work attempts to resolve these issues of ScalaTion by providing a support with distributed databases and microservices to unify multiple modules of the software.

A distributed databases package is designed to support working of the columnar databases over multiple nodes across a distributed system. This package also answers the lack of persistence of relations in databases. A new module `scalation_ms` is introduced which unifies the mathstat, databases and modeling modules. Each service has a specific role and can be set up on different systems.

Section 3 talks on the distributed toolkit used in achieving this work. Section 4 talks more on the work done to provide the above-discussed topics.

3 Distributed Toolkit

At the start of this work, the option of using an existing toolkit that provides distributed functionality was easy to take, rather than doing a system level programming, which is not the main aim of the software. Considering the number of toolkits available and their wide usages, a short study is done on a few prominent ones like Akka [1], Kafka [5], and Netty [6]. Based on the available features per needs, support to the system and ease of use, Akka toolkit is chosen to integrate with the software.

Akka is a toolkit for building distributed systems, written in Scala and developed by Lightbend, first released in 2009. It implements the Actor-model on JVM and can be used for projects written in Java and Scala.

Akka supports building a router system between multiple nodes and also a cluster between the nodes. The cluster model can be used in the microservices system with each node providing certain set of functionalities. With nodes acting as actors, interaction between actors is done by message passing. There can also be multiple actors on one node. Messages sent between actors are mapped by pattern matching. Akka also provides support to persistence models. Akka `remote` dependencies provide a connection with remote systems. This enables setting up the services on multiple systems.

Considering all the available features, Akka stands as an all-in-one support toolkit for the requirement related to this project.

Talking about working with Akka, each class in the actor-model is defined as an actor. A user initiates an `ActorSystem` and requests an `actorRef` to the master node of the

system. The user can interact with the system using this `actorRef`. Messages of actors are defined as `case class` and data is passed in parameters of these classes. A message can be sent to an `actorRef` using `tell` or `!` method. An example of connection and message passing is shown below,

```
val system = ActorSystem ("Microservices Application")
val aref = system.actorOf (Props["ActorClass"], "master-actor")
aref ! print("hello world!")
```

In the above code, `aref` is an actor reference returned by `actorOf` method. It is referencing to `ActorClass` actor. `case class print(s: String)` is the messsage defined `object ActorClass` whose definition is written in `ActorClass` which prints the string s.

4 Procedures

4.1 Distributed Database

The monolithic columnar-database that is currently supported by ScalaTion is not favorable when working with large datasets. It makes the queries run in sequential order leading to queries not related upon one other still having to wait for the previous queries to complete the execution. This can be overcome to an extent with a distributive architecture. It also provides the possibility of distributing the task among multiple systems so that not a single machine has all the burden of huge operations.

The distributed database package is an implementation of Akka cluster, where a master actor called `RelDBMaster` initiates a connection to n worker actors called `RelDBWorker`. An initial implementation using Akka routers was deprecated due to limitations on selecting a certain worker node for operations. The cluster is set up by the master node by creating n connections to worker nodes that are stored in an array. Each worker node is denoted by the index of this array. A map named `relNodeMap: Map[String, Int]` is maintained to keep the track of the relations created on worker nodes, with map entries being `relationName -> nodeIndex`.

A user can open an `ActorSystem` and get a reference to the master actor. And can interact with the database system using the actor reference by message passing. This distributed database system is designed in such a way that, it favors if the dataset is divided into multiple files on local file system which are loaded to create relations. The model of the system is as follows:

4.1.1 Creating relations

Like mentioned, the system is favored if the dataset is divided into multiple *.csv* files. All the files of same relation can be loaded into the system to create the databases by passing the file names in a single message to the master actor. The message to create the relations is

```
createFromCSV (fname, name, colName, key, domain, skip, eSep)
```

where `fname` is a sequence of file names, `name` is a sequence of names of relations of how each of these files will be named as. The option of passing multiple files in a single message is provided to reduce the number of calls to the actor, and to fairly distribute the relations across multiple worker nodes.

Of multiple files sent, each file is sent to different worker nodes in a loop iterating sequentially. These files are thus loaded as relations on worker nodes by calling the

```
Relation.apply (filename, name, colName, key, domain, skip, eSep)
```

If the number of files passed as arguments are more than the number of worker nodes, the cycle of where the relations will be created starts back from node 0.

4.1.2 Relalgebra operations

Now, that all the relations are created on worker nodes, when the user interacts with master requesting for a particular query, master has to co-ordinate with worker nodes to perform the tasks. For most of the operations, a list of names of relations is supported to avoid multiple calls of similar operation. Considering `select` operation, the message call from user looks

like this:

```
select (name, rName, p)
```

where, `name`: sequence of strings - names of relations on which the select is performed. `rName`: sequence of strings - names with which corresponding result relations are named as. `p` is the predicate condition for the select operation.

When the user sends this message to master, master loops through the `name` list to find respective worker node ids using `relNodeMap` and sends a message to the worker to perform select operation on that relation.

```
worker(i) ! selectIn (name(i), rName(i), p)
```

Messages that end with $-In$ are accepted only by the worker nodes. Other operations that work in simlar to `select` are:

- `project` - Takes relation names, result names, column names to be projected on.
- `minus` - Takes two sequences of relation names, result names
- `product` - Takes two sequences of relation names, result names
- `join` - Takes two sequences of relation names, result names
- `delete` - Takes a sequence of relation names

For `minus, product, join`, the data is assumed to be arranged in the worker nodes such that columns on which these operations are performed are aligned on same worker nodes. E.g. if working on a time-series data with $traffic$ and $weather$ information, and both have a $time$ column, the division of the dataset should be done based on the $time$.

To display certain relation as one, (say $traffic$ data is divided into $\{traffic1, traffic2, traffic3\}$, `show` operation can be used by passing the list of sub-relations. When master receives this message, it requests each relation table from the corresponding worker node based on the relation name to receive at the master. The master then performs a union of all these relations once all the requested tables are received, and displays as $traffic$ relation.

For `union` operation, which needs to span over multiple worker nodes to fetch the data and append together, works in a similar fashion as `show` operation. After performing union of tables on master node, a random worker node is then chosen and the result relation of union is sent to that node.

4.1.3 Persistence

The previous version of ScalaTion was lacking with persistence of data. This is introduced in distributed database. Akka persistence provides plugins to store data in in-memory heap based journal, local file-system based snapshot store and LevelDB based jornal. The LevelDB based approach is implemented as part of this project. It works with the default java serializer. The addition of persistence saves time in loading relations from huge files everytime an operation has to be run. Since the relations are created on worker nodes, the data is also persisted on the machines on which these worker nodes are setup. The following persistence methods are available:

- **saveRelation** - save relations to persistence memory
- **dropRelation** - drop relations from persistence memory
- **getRelation** - fetch relations from persistence memory

The working of these methods goes similar to other relation algebra operations.

4.2 Microservices

Microservies are built on Akka clusters with a master having independent connections with worker nodes. Each worker node represents a category of operations inspired from [8]. Currently, this module includes three microservices - database, preprocessing, analytics.

4.2.1 Database

This is a replica of distributed database 4.1. The master actor of the cluster acts as master node of distributed database system connected to multiple database worker nodes. All the

operations available in distributed database are still available here including persistence of databases.

4.2.2 Preprocessing

Preprocessing service includes methods that perform actions on relations, matrices or vectors. The master of cluster is connected with a single connection to preprocessing worker node. A message that belongs to be performed on this worker is forwarded by the master. The list of methods in this service can be found from Section 3.2 in [8]. Typical preprocessing operations include replacing missing values, converting string to numeric columns, outliers and imputation. A single worker node is considered enough for this service because these methods are not huge and are mostly easily performed on a single node better than distributing among multiple nodes.

4.2.3 Analytics

This service is from the modeling module of the system. A connection from master node of microservices is created to the `AnalyticsWorker` node. For the query on master related to analytics operations, they are forwarded to this worker node. With a large number of available methods in analytics, simulation topics, currently only few are added to this service. Abstract classes and traits of the module are used in building the messages of this service.

Classifier classes are built on abstract classes `analytics.classifier.ClassifierInt` and `analytics.classifier.ClassifierReal` and an `analytics.classfier.Classifier` trait. These are used in creating `classify, crossValidate, test, featureSelection, calcCorrelation` methods on various classifiers. The `ClassifierInt` class implements `BayesClassifier, DecisionTreeID3, NullModel` classifiers, while `ClassifierReal` implements `DecisionTreeC45, KNN, LDA, LogisticRegression, NaiveBayes, RandomForest, SVM` along with few other models.

In prediction models, `analytics.PredictorVec` and `analytics.PredictorMat` abstract classes are used to implement various prediction models. `PredictorVec` is used in `ANOVA1,`

`PolyRegression, TrigRegression` and `PredictorMat` is used in `Regression, Perceptron, SimpleRegression, NonLinearRegression, LassoRegression` along with other regrssion techniques.

5 Future Work

- As from the work mentioned above, not all the relation or modeling methods are added to respective microservices. The analytics and database methods support in the distributed system can be expanded.
- A single instance of remote node is implemented, which does not really provide a complete protection of data in case of node-failures. This can be answered by using multi-node replication of data.
- In case of node failures, master node has no idea on how to fix the issue. May be can look on working with deployment management systems like Kubernetes [4] to provide a safe distributed model and ease replication on multiple nodes.

6 Acknowledgement

I would like to thank my major professor Dr. John A. Miller. He has been very supportive throughout my study at UGA. Working with him inspired me to bring the best efforts towards my work and will carry with me forever. I would like to thank all the people I have worked with in the lab (Vishnu, Supriya, Yang, Santosh, Hao, Chandana, Vamsi). I've enjoyed working with each of them and have learnt a lot in the process. Special thanks to Shivani Khatu whose presence is always a roller coaster ride that I enjoy.

References

[1] J Bonér, V Klang, R Kuhn, P Nordwal, B Antonsson, and E Varga. Akka scala documentation. In *Tech. rep.* Typesafe Inc., 2014.

[2] Namiot Dmitry and Sneps-Sneppe Manfred. On micro-services architecture. *International Journal of Open Information Technologies*, 2(9), 2014.

[3] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. Microservices: yesterday, today, and tomorrow. In *Present and Ulterior Software Engineering*, pages 195–216. Springer, 2017.

[4] Kelsey Hightower, Brendan Burns, and Joe Beda. *Kubernetes: Up and Running: Dive Into the Future of Infrastructure.* " O'Reilly Media, Inc.", 2017.

[5] Apache Kafka. A high-throughput, distributed messaging system. *URL: kafka. apache. org as of*, 5(1), 2014.

[6] Norman Maurer and Marvin Wolfthal. *Netty in Action.* Manning Publications, 2016.

[7] John A Miller. ScalaTion - an open source big data framework.

[8] John A Miller. Introduction to data science using scalation. 2018.

[9] M Tamer Özsu and Patrick Valduriez. *Principles of distributed database systems.* Springer Science & Business Media, 2011.

[10] Wikipedia contributors. Distributed database — Wikipedia, the free encyclopedia, 2018. [Online; accessed 13-December-2018].