Final Design Document

Important Note: In EZstack, "documents" can be thought of as rows in a conventional sql table.

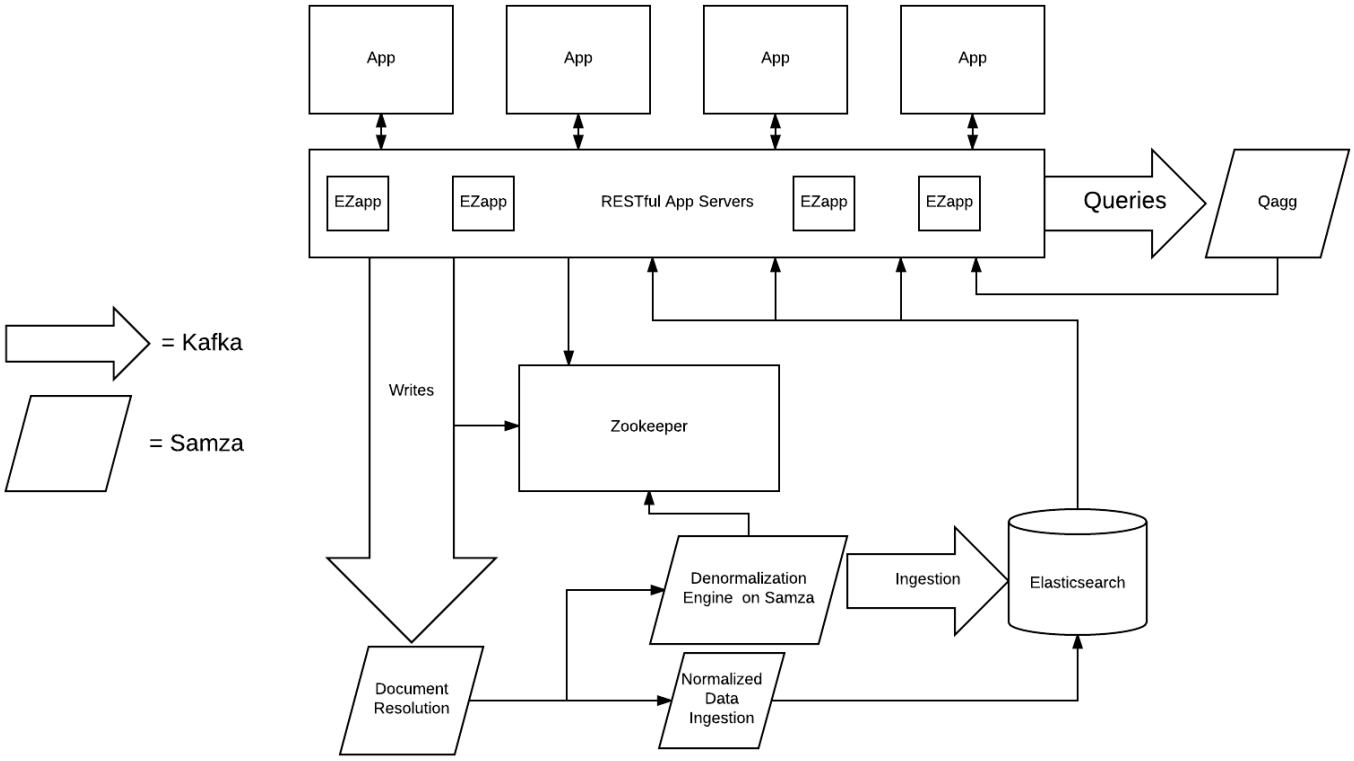
The goal of the project is to provide a highly scalable, highly efficient, eventually consistent data stack. This product is designed to benefit big-data companies in achieving an efficient, highly scalable data processing system with a simple interface, thus eliminating engineering difficulties within the data model. While most other big data stacks attempt to tackle these problems by investing large amounts of resources in developing their own custom solutions that are highly specific to their case, EZstack will attempt to provide a general solution that is applicable to many different use-cases, favoring ease-of-use and customization.

The way that users will interact with the system will be through a RESTful API that enables users to make calls that will perform reads or writes to the system. This will allow many different types of applications and services to run off of this platform.

There are many different use cases for the system, which stems from the fact that there is a very pervasive problem with data intensive applications: their backends are very complex, to the point where engineering them can become nearly impossible. This causes many potentially amazing applications to be very expensive to deploy and maintain. Those extreme expenses make it prohibitive for groups of people that do not have sufficient funds to break into the global software market. Additionally, although large corporations might have the capital required to engineer data intensive applications, there still exists many major problems that they cannot avoid. The most significant of these problems is that many of the existing systems are inflexible. It is nearly impossible to make alterations to an existing stack because most major alterations require the system to be almost entirely re-engineered, which is extremely expensive.

Major Components of the Project

The different components of the project are made in reference to this diagram of components, where you can see how the different parts of the project interact with each other in this specific hierarchy:



Restful API:

The restful api is the main source of interaction within EZstack. The below reference is our preliminary restful interface.

Stability	URL	HTTP Method Type	Body	Functionality
Stable	/sor/1/{table}/{key}	POST	Expects json object. { … }	Creates new document or Overwrites an existing document
Under development	/sor/1/{table}	POST	Expects json object. { … }	Same as above but key is automatically generated.
Stable	/sor/1/{table}/{key}	PUT	Expects json object.	Updates existing document

			{ … }	matching this key or Creates new document if it does not exist.
Under development	/sor/1/{table}	PUT	Expects json object. { … }	Same as above but key is automatically generated.
Stable	/sor/1/{table}/{key}	GET	Returns json object. { … }	Returns the existing object matching this key or an empty object.
Under development	/sor/1/tables	GET	Returns json object. […]	Returns a list of tables.
Under development	/sor/1/search	GET	Experimental Query language example json: Refer to “Query Language Syntax” section below.	Returns all documents matching search query.

Query Language Syntax:

The query language syntax is currently under development and will most likely change. A preliminary example of the query language syntax is provided below.

Example 1

```
{
…
“Query” : {
                “tableName” : “name”,
                “Filter” : (...),
                “Join” : {
                        “Query” : (...),
                        “Replace” :  [
```

```
                                        {
                                                “outerTableField” : “attributeName”,
                                                “innerTableField” : “attributeName”
                                        }]
                        }
                }
}
```

Example 2:

```
{
…
“Query” : {
        “Field” : “fieldName”,
        “Value” : “some value”,
        “Op” : “eq”,
        “Condition” : {
                “conditionOp” : “AND”,
                “Query” : {...}
                }
        }
}
```

Qagg:

The Qagg is EZstack’s query aggregator. It is our algorithmic methodology for benchmarking user queries. Dropwizard metrics for our restful interface are pumped into Datadog, as we currently utilize Datadog for query metrics ingestion. After enough metrics have been ingested, determinations are made on what rules should be implemented in the denormalizer to best maximize the efficiency of the reads. This is accomplished by an algorithm that takes into account the different response times at varying percentiles of each query, as well as what types of increases in efficiency are expected to be able to be made with the rules available on each query.

Denormalizer:

The denormalizer is EZstack’s data manipulation component. The denormalizer adds extra metadata fields to each record inserted for EZstack analytics. These can be seen in the metadata table provided below. The denormalizer also coordinates with the Qagg over Zookeeper to implement data manipulation based on a set of rules. These data manipulation rules are the core of EZstack performance gains. By manipulating the data to naturally fit the user queries we eliminate the need for multiple database/table reads since the data is now compacted into a single record. It also significantly reduces processor computation time

because the data is naturally computed properly at insertion time only once and referenced every time needed rather than being recomputed after every query. A preliminary set of rules can be viewed in the rules table below. The metadata and rules references provided below are in beta and are subject to change as EZstack matures.

Metadata Table:

Metadata fields	Functionality
key	The unique identifier for a column. These keys are globally unique across the entire system.
lastupdated	A timestamp of the last change committed to a document.
version	Version keeps a count of how many document revisions have occurred to the document.
createdAt	Timestamp of the first appearance of the document in our datastore.

Rules Table:

count	Provides a counter for a specific document field attribute.
join	Joins two types of documents into a new document.
max	Provides a max value for a specific document field attribute.
min	Provides a min value for a specific document field attribute.
sum	Provides a sum value for a specific document field attribute.
average	Provides an average value for a specific document field attribute.
expand	Assuming a document contains an attribute referencing another document. Expand will create a new document that will replace that

	field attribute with the actual document that was being referenced by it.

Zookeeper:

EZstack leverages Kafka and Samza in our data store, both of which rely heavily on Zookeeper for coordination across the clustered infrastructure. The Qagg also relies on Zookeeper to communicate the rules needed for the denormalizer to denormalize data. EZstack agents also might utilize Zookeeper for coordinating the different components that build up EZstack.

DataWriter:

EZstack is built as a series of components. It relies on interfaces for underlying infrastructure abstraction. The default EZstack datawriter is Kafka. Data is first written to Kafka, and over the span of the system it will reach the denormalizer through our document resolution component of the system. To change the default datawriter system from Kafka to another system of your preference the following interface below needs to be implemented for that system. It is highly recommended that the developer should fully understand the pros and cons of moving away from Kafka as the datawriter system to another preliminary writer system.

DataWriter.java

```
public interface DataWriter {
	void create(String database, String table, String key, Map<String, Object> document);

	void update(String database, String table, String key, Map<String, Object> update);
}
```

DataReader:

EZstack also implements a similar interface in the DataWriter for data reading. The default underlying system that implements the DataReader is Elasticsearch. Our system decouples reads and writes because of the denormalizer component, and while at first glance it might seem counterintuitive to have two different systems to handle the reads and writes, it becomes much clearer after realizing that all data is processed through the denormalizer first.

DataReader.java

```
public interface DataReader {
	Map<String, Object> getDocument(String database, String table, String id);

	List<Map<String, Object>> getDocuments(Query query);
```

}

How Data Moves Through EZstack:

The following will be a high level overview of how data is transported through the different stages of our system. This assumes that Kafka is the DataWriter and ElasticSearch is the DataReader. If the developer has implemented the interface for a different system, simply remove the Kafka or ElasticSearch phrasing with the replacement system's name.

Users interact with EZstack exclusively through the EZstack RESTful interface. While users can directly interact with ElasticSearch or Kafka, that is not supported by EZstack and might result in data inconsistencies.

What Happens on Data Write:
On data write the RESTful api takes the user input document and writes it to Kafka. After the document has been written to Kafka, the document resolution component determines if the data needs to be denormalized or inserted in a normalized form. All data is always inserted in normalized form, and then based on the set of rules provided by the Qagg the data might also be processed through the denormalizer component as well. Regardless of whether the data is transmitted through the normalized or denormalized component the data will be produced into a secondary Kafka stream that will eventually get flushed into ElasticSearch. Please note that data travels through a series of streams before becoming available to read, and it should be very clear that EZstack is an eventually consistent data model and should not be used to replace transactional systems needs.

What Happens on Data Read:
On data read the RESTful api takes the user input query and search for relating data in Elasticsearch. If results appear in Elasticsearch then they will be returned to the user. While the RESTful api searches for the result, the query statistics are sent to the Qagg for analysis. The Qagg analyzes the query and determines if new rules need to be generated.

Frequently Asked Questions:

How does the RESTful api read the denormalized data?

The RESTful api utilizes the query class to determine if there is a rule mapping to that query, and if so, it will query the data from the newly created table for that rule.

How does the document resolution component determine if data should be sent to the denormalizer or the normalizer component?

The data processed by the document resolution is compared against the rules generated by the Qagg in Zookeeper. If the data matches any of the rules then it is transmitted to the the

denormalizer, otherwise it is transmitted to the normalizer. Also, if the data doesn't already have our metadata included then the data is being produced directly from the RESTful api and it is sent to the normalizer automatically.

Timeline

The project's timeline is quite well defined for the beginning of the project's development. There are certain objectives that simply must be met for everything to get off the ground and running in a sufficient manner. Early on, there will be a lot of testing that will go into using the different technologies involved in the project. Simply getting different technologies to work in the necessary manners will take some amount of work, and although not the most difficult area of the project, it still is one of the most important milestones of the project to have individual technologies working on their own.

The first large goal of the project is to have a very simplistic working version of EZstack, where the parts of the project are put together, with the only pieces not included being the ones that add the significant complexity to the project. This means that the project at this point in time would essentially consist of a pipeline framework capable of moving written data through Kafka streams and Samza jobs into an Elasticsearch engine, where it is possible to manually query the objects. This will be a very significant challenge, because despite the pieces being able to work flawlessly on their own, getting them to work well together will require a large amount of work. Among the problems involved will be deciding on the versioning of each platform, which will dictate what work is necessary to put the pieces together in the first place. However, this will also involve determining what external technologies will be necessary to get everything working together as well. This means a large investigation into Apache Zookeeper, as this will be the technology used to get the different components of the project coordinated together, and give them the ability to communicate with each other efficiently.

The working version of the project in this state will not have the efficiency that EZstack strives to achieve. Instead, it will be more of a proof of concept that the system is capable of functioning at a basic level, and from that point on, the goal will be to improve on the designs to the point where we can make progress towards having a product that is actually desirable.

Once the project is at a state where there is a working pipeline, the goal will be to iterate over the project, adding the required complexities to give the project its intelligence. This means the next major milestone will be having a basic working query aggregation system that can successfully determine and communicate specific rules that need to be made by the denormalization engine, based on user queries sent by the RESTful app server to the query aggregation system. The intention for this milestone would be to have a system that provides a satisfactory level of efficiency to the point where it can be determined that the project is successful, but isn't to the point where the project has been finalized, as there will still be improvements that are possible to be made in a short-term time period. The working version of

this will have at least a level of efficiency that EZstack guarantees for the final project, with the capability of further improvement in the future.