etcd
A key-value NoSQL solution for distributed systems

FÁBIO SÁ, FEUP, Portugal
INÊS GASPAR, FEUP, Portugal
JOSÉ A. GASPAR, FEUP, Portugal

The present report aims to explore, both theoretically and practically, the *etcd* technology, a non-relational database using key-value paradigm. Throughout the document, characteristics of the technology are presented, as well as its specificities and use cases. Next, a specific scenario is addressed, along with the strategies supporting the solution found. Finally, the implementation of the prototype using *etcd* is explained, considering the aforementioned ideation, *etcd* characteristics and some considerations of features in this paradigm.

Additional Key Words and Phrases: NoSQL, *etcd*, Key-Value, consistency, distributed systems, full replicated

ACM Reference Format:
Fábio Sá, Inês Gaspar, and José A. Gaspar. 2024. *etcd* A key-value NoSQL solution for distributed systems . 1, 1 (May 2024), 17 pages.

1 INTRODUCTION

The increase in the amount of data and the complexity of systems has led to the creation of new database solutions, non-relational databases [37], with the aim of providing efficient storage and retrieval of data and allowing rapid access in large-scale applications.
In this way, various paradigms have emerged, including key-value, which is explored in this report. This approach has a very simple design, since it only depends on the design of the key and most of the processing and manipulation required is on the application side. These NoSQL technologies allow for great scalability and better time performance than relational databases. They are therefore often used as caching systems, where data is temporarily stored for quick access. However, the use of these strategies can lead to a problem known as *Impedance Mismatch* [36], i.e. discrepancies between the models of the structure of the technology used in the application and the model used in the database. *etcd* is a database often used for setting up distributed systems. It is especially used because of the strong consistency between nodes that it allows, which will be explored in the following sections, as well as through the use cases of this technology and the prototype developed.

2 TECHNOLOGY

In this section, *etcd* is going to be described in terms of features, data model, advantages, limitations an some use cases are also presented.

2.1 Overview

etcd is a distributed key-value database [2]. Key-value databases are a type of NoSQL database paradigm that stores data as a collection of key-value pairs, where each key is unique and associated with a single value. This database system was initially developed by CoreOS [20] in 2013, when it had its first release. In 2018, RedHat [42] announced the acquisition of CoreOS, and IBM [32] announced the acquisition of RedHat in the same year. *etcd* is free and follows an open-source licensing model. Its official documentation [23] features many tutorials, demos, and installation instructions, as well as an extensive FAQ [8]. The community [16] is active and supportive, with user and developer forums on Google Groups [18], real-time updates on Twitter [19], and discussions on GitHub [17]. Additionally, contributors and maintainers

Authors' Contact Information: Fábio Sá, up202007658@fe.up.pt, FEUP, Porto, Portugal; Inês Gaspar, up202007210@fe.up.pt, FEUP, Porto, Portugal; José A. Gaspar, up202008561@fe.up.pt, FEUP, Porto, Portugal.

Manuscript submitted to ACM

hold weekly online meetings via Zoom, with meeting documentation available and sessions archived on YouTube. As of the date of this report, it ranks 54th among the most used database engines and 5th in the key-value paradigm, according to the evaluation on db-engines.com [25].

The name "etcd" originated from two ideas: the Unix "/etc" folder and "d"istributed systems [39]. The "/etc" folder is a place to store configuration data for a single system, whereas *etcd* stores configuration information for large-scale distributed systems. Thus, it is widely used for configuration management, service discovery, and coordination in distributed systems.

etcd provides a reliable way to store data across a cluster of machines and ensures strong consistency guarantees. Unlike most other NoSQL databases, *etcd* is designed to be highly available and fault-tolerant, making it a popular choice for storing data in distributed systems [32] [28]. As specified by the CAP theorem [33] (Consistency, Availability, Partition Tolerance), in order to achieve strong consistency, *etcd* sacrifices performance, which makes it not ideal for projects where execution time is crucial.

2.2 ACID Properties

etcd guarantees the ACID properties [29] (Atomicity, Consistency, Isolation, and Durability) within transactions. However, they only exist for one key-value set, i.e. it is not possible to change more than one key-value pair in a transaction [15]. A transaction is an atomic If/Then/Else construction on key-value pairs. They are often used to protect keys from unintentional concurrent updates, create compare and exchange operations and develop better higher-level concurrency control. They can atomically process several requests in just one. Each change to a key-value pair causes the revision (the version of a given key-value pair) to be added for the given transaction, as well as all the events triggered by the transaction being part of the same revision. However, it is not allowed to change the same key several times in a single transaction. Transactions are protected by comparisons, i.e. each of these comparisons checks a single key from those stored. In this way, they work by checking the absence or presence of a value and comparing it with a given one, or by checking the revisions. In terms of comparisons, two different ones can be applied to the same key or to different keys. These are applied atomically as follow: if all the comparisons are true, then the transaction was successful and *etcd* executes the then (success) request block for the transaction, otherwise it is said to have failed and the else (failure) request block is executed.

2.3 Features

As it is going to be described throughout this section, *etcd* has several features, some of them unique, that make this database a good choice in many distributed systems.

2.3.1 Replication and node communication features.

This database is built to work in a distributed way [6], when the number of nodes / machines is greater than one. This cluster can be generated through an internal network between several nodes. In *etcd* the number of nodes is preferably odd for resource management purposes.

etcd is built on the Raft consensus algorithm [1] to ensure data store consistency across all nodes in a cluster for a fault-tolerant distributed system. This algorithm is based on quorums and as the name suggests it is used to have a consensus between a majority of nodes about the values that are being stored in the database, taking into account that one or more nodes may fail. In *etcd*, for a cluster with N members, the quorum size is (N/2)+1. For any odd-sized cluster, adding one node will always increase the number of nodes necessary for quorum.

These nodes do not need to be physically together, even though it may affect request latency. *etcd* stablishes connection between nodes via HTTP + TLS [3]. If the IP of the node is not known, *etcd* has a discovery mode that will find the node's IP address and establish the connection. In terms of load balancing, it is usefull to state that there is a leader node. This node is responsible for ensuring data replication among non-leader nodes and balance the distribution of request among those nodes [8]. Even though it is possible to have multiple nodes in one cluster, *etcd* does not provide a way to support multiple clusters that can communicate with each other. To implement that feature, some communication protocol must be implemented between the clusters. One possible approach would be to put the leader node of each cluster in charge of that communication.

Each node has a copy of the data, and the data is replicated across the cluster. This means that, if a node fails, the data is still available on the other nodes, even if the leader is down - reelection is done. As such, having a leader does not represent a critical point of failure for the system. It only fails if the majority of nodes are down, that is, consensus is not obtained. In *etcd* there is a total replication of the data between all nodes, which means, it's full replicated.

2.3.2 Consistency features.

etcd provides sequential consistency, which is the stronger form of consistency that can be obtained in distributed systems. This means that, independently of the node of the cluster that receives the request from the client, it reads the same events in the same order.

Note that eventual consistency is not enough, specially in critical systems where, if in any moment, there are inconsistent states on the nodes, since they probably have configurations of systems stored in it - main purpose of *etcd* - it can cause critical problems to the systems, making it, for example, vulnerable to some mallicious attacks.

It's also important to mention that, each key has a version that is required internally to achieve consistency in the distributed system and *etcd* allows users to see those versions and the respective values.

Consistency is one of the advantages of using a distributed database. It allows for multiple nodes to be updated at the same time, which can lead to inconsistencies in the data. To avoid this, *etcd* provides a quorum like strategy, which ensures that the data is consistent across all nodes in the cluster even if some nodes are down at that given moment[12].

2.3.3 Watcher feature.

etcd provides a functionality called Watcher [10]. This watcher can be used to monitor a given value of a certain key over time based on the operations executed over that key-value pair.

It's possible to specify whether to monitor only the PUT, only the GET operations, or both, depending on the problem. With this feature it can be seen how useful this database is regarding system configurations. With this monitoring feature it is extremely easy to see modifications in critical variables in real-time (without resort to polling), helping to prevent any unwanted results.

2.3.4 Data processing features.

Regarding data processing features, *etcd* provides a limited range of those. As an example, functions like counts, sums, averages, map-reduces that are supported by other databases have no translation in *etcd*, making mandatory to process information after querying the database or designing key-value structures in a way to make post-processing unnecessary. As it can be seen in the oficial documentation of *etcd*, the main supported features are methods to read, write and delete data, besides the ability to monitor changes in a given key-value pair plus the possibility of knowing the version of the key and seeing old values for a given key [11] [12].

However, it is not only in data processing features that *etcd* is not ideal, also on the data types that can be stored in *etcd*. These types consist in strings and numbers, so there are no lists, sets or more complex data types.

2.3.5 Prefix feature.

etcd has another peculiar feature that can be very handy, the prefix search. This feature allows users to search for a range of keys based on the prefix. Since *etcd* is fully replicated, it is possible to obtain all the keys with the same prefix only performing one request to exactly one node of the cluster, making this operation efficient. To exemplify this feature, if the keys **app:foo** and **app:bar** exist, the prefix **app:** can be used to retrieve both keys and their respective values.

2.4 CLI and Client libraries

etcd provides a CLI, etcdctl [22]. Currently in version 3, this command line interface allows interacting with the database at a lower level. It's a wrapper for the documented API.

Due to its widespread use, *etcd* is combined with various open-source libraries [24], ensuring a suitable interface between *etcd* clusters and the backend of applications that use this technology. Examples include Microsoft's etcd3 [34], suitable for Javascript and Typescript, and Python-etcd3 [45] for Python.

2.5 Data Model

The data model [7] can be seen logically and physically. Throughout this section both of them are going to be described.

2.5.1 Logical View.

The database's logical view is a flat binary key space with a lexically sorted index for efficient range queries. It maintains multiple revisions, with each atomic mutative operation creating a new revision. Old revisions of keys remain accessible through previous revisions, and they are indexed for efficient ranging. Revisions are monotonically increasing over time.

A key's life spans a generation from creation to deletion, with each key having one or multiple generations. Creating a key increments its version, starting at 1 if it doesn't exist. Deleting a key generates a tombstone, resetting its version to 0. Each modification increments a key's version within its generation.

2.5.2 Physiscal View.

etcd stores data in a persistent B-Tree [30], with each revision containing only the delta from the previous one for efficiency reasons. Keys are represented as 3-tuples (major, sub, type), allowing differentiation and optional special values like tombstones. The B-Tree is ordered lexically for fast ranged lookups over revision deltas. Compaction removes outdated key-value pairs. Additionally, *etcd* maintains an in-memory B-Tree index for speedy range queries, with keys exposed to users and pointers to modifications in the persistent B-Tree.

2.6 Supported Data Operations

etcd provides an HTTP/JSON API [12] that allows clients to perform CRUD operations [4] on the database. There are mainly 2 operations, GET and PUT. As the names suggest, these operations are used to retrieve a given value based on a key and store a new or an existent key-value pair, respectively. Regarding updates, they are just a new PUT, over the same key. It is also possible to delete key-value pairs using the delete operation. Another interesting feature provided by the API is the ability to check the cluster' health. Parameters like the leader node and the number of requests can be seen via this API. Regarding the watchers, via the API, it is possible to create them and generate a function that is going to run at each operation over the key that is being monitored [12].

2.7 Use Cases

As previously mentioned, *etcd* is used mainly to perform system configurations in distributed systems [14]. The most important use cases are:

- Kubernetes [40] - Uses *etcd* to store all its data – its configuration data, its state, and its metadata.
- Container Linux by CoreOS [31] - Uses *etcd* to store the configuration data of their internal distributed system.

2.8 Problematic Scenarios

One of the problematic scenarios of *etcd* is related with the restricted data types that can be stored in the database. Only numbers and strings are accepted. There is a need to keep a json encoded as string when more complex types are required and the attribute search over the value is impossible: the system must receive the entire entity and then select the attribute manually.

etcd was developed to deal with small key-value pairs, typically designed to metadata. Hence, the maximum size of any request is 1.5 MiB. Similarly, the suggested maximum size of the database is 8 GiB (2 GiB is the default). This database was not built to be used as a cache, as most key-value databases[13], since the data is allocated in the hard-disk.

Fast disks are vital for *etcd* performance and stability [9]. Slow disks increase request latency and cluster instability. *etcd*'s consensus protocol requires timely storage of metadata to a log, with most cluster members writing every request to disk. *etcd* also checkpoints its state to disk for log truncation, and delays in these writes can cause heartbeat timeouts, triggering cluster elections and instability. *etcd* is highly sensitive to disk write latency, needing at least 50 sequential IOPS (e.g., from a 7200 RPM disk) and ideally 500 sequential IOPS (e.g., from a local SSD or high-performance virtualized block device) for heavily loaded clusters. This limitation, together with the inability of writting a block of

key-value pairs in one operation - would eventually surpass the 1.5MiB of request - makes the entire cluster slower when facing lots of consecutive writes. This drawback is noticeable when populating an *etcd* database for example.

Furthermore, *etcd* can also become problematic due to its own nature of replicating data totally. It is possible to have several nodes running to ensure data is (almost) never lost, however this makes the entire system slower with the increase of nodes, precisely due to data consistency and replication.

2.9 *etcd* vs. Other Solutions

As previously mentioned, *etcd* is not like most key-value databases. It is recommended to be used when sequential consistency is needed, in distributed systems, and not to be used as a cache, like Redis [43], for example. There is a tradeoff between having a fast response time and having a consistent state. *etcd* is made not to be fast but consistent while most key-value databases are made to be fast but not consistent. The main advantage of *etcd* is the ability of having more than one point of failure. Thanks to the Raft consensus algorithm, if more than 50% of the nodes are up and running, the system will be able to continue working and accept new operations over the database. Then, when the nodes that fail recover, the state is fully replicated to those nodes achieving the sequential consistency that is needed and guaranteed by the API. Other advantage, as specified in *etcd* documentation, is the maximum reliable database size.*etcd* can scale up to several gigabytes while others (e.g ZooKeeper [27] and Consul [5]) can only support scalable results until hundreds of megabytes. The main drawback is the performance of *etcd* and the lack of ability of inserting larges amount of data in a block. Operations on database are restrict to one at a time.

3 PROTOTYPE

To validate the main features, qualities, and potential bottlenecks of the mentioned *etcd* technology, a prototype — proof of concept based on this key-value paradigm — was developed and presented in the following sections.

3.1 Topic

In a classic scenario of purchasing tickets for events, data consistency is the most important aspect to consider. For example, each event should display its updated information in real-time, including the number of remaining tickets, regardless of the number of simultaneous purchases occurring. A failure in data replication and propagation within the system can result in event overbooking, inaccurately computed purchases, and potential revenue loss.

To ensure the system is immune to such issues, it is necessary to rely on a database that is simultaneously:

- Distributed, for horizontal scalability ensuring the system can handle the growing workload effectively;
- Fully replicated, where every data read returns the latest data write across all clusters and nodes, something that is not achievable with eventual consistency;
- Highly available, to have no single point of failure and gracefully tolerate hardware failures and network partitions.

TickETCD, a web application for purchasing tickets for events, uses *etcd* as a solution to the aforementioned problems. The implementation details will be described in the following subsections.

3.2 Dataset

Given the unavailability of data of this nature suitable for the application, TickETCD's data is generated using the Python Faker Library [41] and a configuration file. Among other factors, this stage allows for changing the number of created users, the number of events, the probability of a user being an administrator, the probability of creating and triggering a notification, choosing locations and event types, as well as ticket types and price ranges.

These configurations are important to establish a system governed by scalable and parameterizable data. Although the data is invented and probabilistic, it is also reliable, adapted, and aligned with reality.

3.3 Conceptual Data Model

In order to meet the needs of the application, the following relationships have been designed, as shown in Figure 1.

An event has a name, location, date, type, description, and a total quantity of available tickets. Each event may offer various ticket types, each associated with a price, as well as their initial and current total quantity. Users, identified by

their name, email, password, and role, can have favorite events and request notifications when the quantity of tickets for a specific event reaches a certain limit. Additionally, users can purchase various types of tickets.

3.4 Physical Data Model and Data Structures

The previous Conceptual Data Model could be implemented physically using relational database schemas, allowing for direct searches for relationships between entities. However, in the case of dealing with the key-value paradigm used in *etcd*, it was necessary to resort to data redundancy to ensure complete knowledge of all relationships and still minimize the number of post-processing steps.

Below are presented the key and value structures and aggregates used for the design of the entire data structure required by TickETCD. It should be noted that for the purpose of data visualization and manipulation, JSON was used, although physically, they are just strings after serialization, as *etcd* does not support other data types as values for its keys.

3.4.1 User.

Without any post-processing, it is possible to query all information related to a user by querying the key in the format **user:<USERNAME>**. Example:

```
"user:johndoe": {
    "name": "jonh doe", "email": "john@mail.com",
    "password": "john123", "role": "admin"
}
```

3.4.2 Event.

Similarly to a user, the information about an event can be accessed by a simple query using the key **event:<ID>**. Example:

```
"event:92fe965d-a189-4f26-844c-0979c6ca035e": {
    "name": "Simple concert",  "description": "A simple event example",
    "location": "Porto", "type": "concert", "date": "2024-03-13",
    "current_quantity": "14"
}
```

3.4.3 Ticket.

Given an event and a ticket type, it is possible to determine their current characteristics by using the key in the format **ticket:<EVENTID>:<TYPE>**. Example:

```
"ticket:92fe965d-a189-4f26-844c-0979c6ca035e:pink": {
    "total_quantity": "34", "current_quantity": "23", "price": "23.99"
}
```

The ticket type is the same and fixed for all events, so there would be no issue in declaring this key in the format **ticket:<TYPE>:<EVENTID>**. The ticket types will be addressed in a later section.

3.4.4 Notification.

Given that a user can activate a notification for a specific event, the structure **notification:<USERNAME>:<EVENTID>** was used to store this data:

```
"notification:johndoe:92fe965d-a189-4f26-844c-0979c6ca035e" : {
    "limit": 42, "active": true
}
```

As defined, and leveraging *etcd*'s feature of searching by key prefix, the system also has direct access to all notifications for a user by searching only for prefix **notification:<USERNAME>**. This way post-processing was avoided.

3.4.5 *Favourite.*

With the key in the format **favorite:<USERNAME>**, a single operation is sufficient to ensure the retrieval of all events marked as favorites by the user:

```
"favourite:johndoe": [ "92fe965d-a189-4f26-844c-0979c6ca035e" ]
```

3.4.6 *Purchase.*

Due to potential key collisions in a distributed context, indexing purchase keys by timestamp became unfeasible. Therefore, the purchase history of a user for an event can be queried using the key **purchase:<USERNAME>:<EVENTID>**. Example:

```
"purchase:johndoe:ad25c85c-6714-4d1f-857b-9bcd1a45ccb9": [ {
        "date": "2024-03-14 13:45:00",
        "tickets": [{ "type": "red", "quantity": "3"}]
} ]
```

A purchase is characterized by an array of transactions, each containing a timestamp and a list of purchased tickets. Since the event ID is already present in the key, redundancy was avoided by not including the event identification again here, as it already contains these tickets.

Just like in the case of notifications, leveraging *etcd*'s feature of searching by key prefix, the system also has direct access to all user purchases by searching only for the prefix **purchase:<USERNAME>**, without requiring post-processing.

3.4.7 *Search.*

One of the features to explore in TickETCD is the search for events by string, type, and location. Since *etcd*, being a key-value database, does not allow searching by values but only by keys, an inverted index was implemented:

```
"search:text:some": [ "92fe965d-a189-4f26-844c-0979c6ca035e" ]
"search:type:concert": [ "f2af5c43-7cad-49f8-88c1-2ff7e8fe8d81" ]
"search:location:lisbon": [ "97636456-a096-4868-9dc1-aac79a22961c" ]
```

As observed, the key is constructed based on the search type followed by the input, in the format **search:<TYPE>:<INPUT>**. Its value is always a list of events that owns these characteristics. This also requires initial processing and runtime processing of the strings that constitute the event, such as the name and description, something to emphasize in the limitations of this implementation.

The text search power *etcd*'s prefix search feature, allowing users to search not only for a single word but also for the prefix of that word and obtain the same results without additional computational cost.

3.4.8 *Static data.*

To ensure and enforce system constraints, some static auxiliary structures have been added to the database. Examples:

```
"event:locations": ["lisbon","porto","braga"],
"event:types": ["concert","theater","dance", "magic","circus"],
"ticket:types": ["pink","blue","green","red"]
```

Event locations, event types, and ticket types are frequently accessed structures, allowing for rapid data selection without the need for complex queries or additional post-processing. However, this adds more redundancy to the system.

3.5 Architecture

The architecture of the prototype can be illustrated according to the schema present in Figure 2.

To simulate a distributed system and evaluate its capabilities, the project deployed a cluster comprising five interconnected *etcd* nodes through an internal network using Docker [26] containers. Furthermore, employing Docker as well, a web application powered by Node.js [38] was created, featuring a frontend crafted using the Tailwind CSS

[44] framework. The Microsoft etcd3 library [24] served as a server-side solution for interfacing the cluster and the web application.

After the dataset specified earlier is generated, there is the step of populating the cluster, which takes the most time. *etcd* does not have the capability to receive data in bulk, so each key-value pair must be injected directly into the cluster independently and sequentially. Since this technology is fully replicated and the prototype requires many auxiliary structures, even with just 10 users and 10 events, it easily scales to around 400 key-value pairs, making the populate step slow.

The system is designed to allow manual querying of information in the database at any given time. Due to the absence of a proprietary querying language, a Python script consuming queries in JSON format is used for this purpose, directly injecting commands into the Docker containers using the command line interface and the HTTP API.

The setup and execution of all steps is aided and automated with the provided makefile.

After the setup is completed, the prototype allows access to various endpoints to perform tasks and test the functionalities related to the *etcd* technology:

- /: Used for login or registration;
- **/home[?search=<INPUT>]**: Homepage. By default, it displays some events. If the user searches (using the search bar), it shows the search results;
- **/admin**: Admin page displaying database cluster statistics, events, and event creation;
- **/profile?username=<USERNAME>**: Displays details of a user profile, favourite events and last purchases;
- **/notifications**: Displays the current user notifications;
- **/event?id=<ID>**: Displays details of an event;
- **/tickets?eventid=<ID>**: Used for purchasing tickets;

Given that the database is a distributed cluster of five nodes, the architecture allows for the shutdown of up to two of these nodes, and the system remains intact and fully functional. This is a feature to be explored in the following sections.

3.6 Features

The features implemented in TickETCD are aimed at exploring *etcd* from a practical standpoint, emphasizing its strengths but also highlighting the weaknesses of certain approaches with this technology.

3.6.1 Data processing and Queries.

TickETCD provides a user registration on the homepage, Figure 3, based on the construction of a key-value pair, where the key follows the format **user:<USERNAME>**, with attributes such as name, email, password, and role added to the value. On the other hand, the login process involves simply a GET request for the possible key, followed by a comparison of the password with the stored attribute if the key exists in the system.

The information displayed on the user profile page, Figure 4, is obtained through three queries: a GET request for the key **user:<USERNAME>** to retrieve the user's personal information, a GET request for the key **favourite:<USERNAME>** to list the user's favorite events, and a GET request for the key **purchase:<USERNAME>** to create the purchase history. None of these three behaviors require any post-processing, thanks to the redundancy added during the initial data modeling.

On the event page, Figure 5, the user can add or remove that event from their list of favorites. Internally, it's just a PUT request to the value of the key in the format **favourite:<USERNAME>**, which is a list containing all events marked as favorites by the selected user. Ticket purchases are made on an auxiliary page, Figure 6, which categorizes the prices and quantities available for each ticket type for a given event, querying the values of keys in the format **event:<EVENTID>:<TYPE>**. Internally, during the purchase, a new entry is added to the array stored in the key **purchase:<USERNAME>:<EVENTID>**, and there's a subsequent update of the current value of available tickets for each type and globally for the event. It's worth noting that operations in the case of purchases should ideally be performed within a transaction for concurrency reasons, but *etcd* doesn't allow transactions manipulating various aggregates, as explored further in the limitations section.

On the notifications page, Figure 7, the user can have a visual feedback of their notifications and which ones are active. It's simply a GET request for the key in the format **notification:<USERNAME>**.

On the admin page, Figure 8, the user can view event statistics, created based on queries and subsequent cumbersome post-processing. It's also possible to create an event, generating internally a unique ID and adding as the value to the key **event:<ID>** the corresponding attributes such as name, location, description, type, and each of the ticket types, including their type, initial quantity, and unit price, Figure 9.

3.6.2 Specific Features.

The event page features a notification creation mechanism that leverages the Watcher feature of *etcd*. Once enabled, the database passively waits for updates to the value of the key **event:<EVENTID>**. The only way to update its value is by decreasing the value of the current quantity attribute through a ticket purchase. *etcd* takes care of notifying the user as soon as this value falls below the chosen threshold.

Despite not allowing searches by attributes, the creation of auxiliary search structures, as seen in the previous section, made it possible to emphasize *etcd*'s key prefix search feature. With this, using auxiliary structures in the form of inverted indices, it was possible to provide event searches in the homepage, Figure 10, based on locations, types, and also the text contained in both their titles and descriptions.

Being suitable for distributed systems, *etcd* provides an API for real-time visualization of the health of the cluster and each of its nodes. A visualizer of this information was implemented on the admin page, Figure 11.

3.7 Limitations

Although *etcd* is suitable for most cases explored in the TickETCD prototype, there are situations where an implementation with another NoSQL paradigm or even a relational mode would be more favorable.

The dataset used is indeed very redundant, ensuring a proper establishment of all the relationships proposed in the Conceptual Data Model and minimizing post-processing by the application, since there are only GET, PUT and DELETE operations. Therefore, even with a reduced number of users, tickets, and events, it easily scales to hundreds of key-value pairs. A relational approach would be much more efficient in terms of space, but it would also require further processing by the application, which is suppressed in the case of key-value pairs with the addition of this redundancy.

The entire dataset must then be fully replicated, impacting the system response time on any request to update values in the database. In case of ticket sales, the processing speed of the system would also be a factor to consider when choosing the technology.

etcd does not have the capability to receive data in bulk, so each key-value pair must be injected directly into the cluster independently and sequentially. Given the exponential growth of key-values, this becomes a slow process. In real-world cases where *etcd* is used, for example in Kubernetes for configuration maintenance, this is not a relevant issue because configurations are mostly finite, small in size, and do not grow much. On the other hand, since TickETCD is a web application for ticket sales management, the initial setup becomes a limiting factor of the system.

Unlike other key-value paradigm technologies, *etcd* does not support additional data types in the value of each key beyond strings. This brought a significant limitation in terms of processing objects and the design of the Physical Data Model itself. There are cases in the system where this difficulty could have been overcome by adding extra redundancy, as explained in the following example:

```
"user:johndoe": {
    "name": "jonh doe",  "email": "john@mail.com",
    "password": "john123",  "role": "admin"
}

"user:johndoe:name": "jonh doe"
"user:johndoe:email": "john@mail.com"
"user:johndoe:password": "john123"
"user:johndoe:role": "admin"
```

This would avoid resorting to serialization and deserialization of objects at runtime, before and after accessing the database. However, the first strategy was adopted in the prototype for two main reasons. On one hand, this example cannot be generalized for all the application's needs. In cases like the Favorite aggregate, for spatial efficiency reasons, the value would have to be an array with the IDs of the favorite events, otherwise we would have a key of the format **favorite:johndoe:<EVENTID>'** with a boolean value for all username-event combinations or only for true combinations. In either case, to discover the set of favorite events for a user, as many queries as events in the system would have to be performed. In the case of creating the Purchase aggregate, it is even more harmful, since it is impossible, in a concurrent context, to create keys based on timestamps due to probable key collisions. On the other hand, the approach with more redundancy is not scalable, as for a simple aggregate with N attributes, it would result in N queries to the system and subsequent processing of its data. Therefore, it is always necessary to resort to auxiliary data structures, such as arrays and objects, to meet the application's needs. Given these constraints of the system, and even though there is no solution capable of addressing the trade-off exposed, the approach with a query and subsequent serialization/deserialization is more suitable for TickETCD.

Still within the realm of efficient computing, like any key-value paradigm database, *etcd* does not allow direct access to the attributes of the manipulated values. This results in a very large overhead when the system needs to compute statistics with those attributes. The statistics feature was developed in TickETCD to illustrate this difficulty. Computing a series of visual statistics about the events such as revenue or sales distribution by ticket type was achieved using a lot of external computation. In a system driven by a relational database with support for traditional SQL and operations like AVG, SUM, the task was performed directly within the database system. As such, searching for attributes of various aggregates in TickETCD was compromised and reduced to a tiny fraction of what could be expected in a ticket sales system.

The full-text search feature for events based on their attributes had to resort to external pre-computation of inverted indexes outside of *etcd* to ensure its correct implementation. This caused extreme overhead in processing the name and description strings of the events, as well as the creation of a number of keys equal to the number of different words found throughout the system. This solution is not viable and could be addressed, for example, by using a relational approach with the LIKE operator or even substring search of attributes offered by most databases in the document-based paradigm.

This was the biggest functionality bottleneck encountered in TickETCD considering the purpose of the application and led to the decision not to implement a way to update event data. In fact, a modification to these attributes would trigger a delete of all references to the event's words in all auxiliary structures and subsequent computation of all affected inverted indexes. This update would cause extreme overhead that was chosen not to be supported in this prototype.

etcd has a significant restriction when it comes to transactions. Transactions are allowed within the same aggregate, meaning within the same key-value pair, but not between different aggregates. In a concurrent scenario where the prototype is involved, the lack of general transactions causes consistency problems during concurrent moments. The most important scenario for TickETCD where this robust feature would be crucial is in the purchase scenario. A purchase triggers a comparison, a possible decrease in the overall number of tickets for the event, and a decrease in the quantity of tickets for each type. These operations, which involve different aggregates, can only be implemented sequentially and not atomically.

4 CONCLUSION

Through the development of this project, the exploration of one of the paradigms of non-relational databases, key-value, was deepened. This provided the opportunity to study a new technology, *etcd*, and understand its specificities and the cases in which this type of tool is most suitable for use, since its entire design is very much focused on system configurations and distributed platforms where it is essential that there is strong consistency between the nodes of the cluster. The implementation of the prototype, in turn, made it possible to understand this approach and to apply features such as the scalability and consistency that these databases offer. The study of these differentiating properties of non-relational databases is fundamental given the growing volume of data and operations in today's systems.

5 ANNEXES

This section shows diagrams and screenshots of the platform developed:

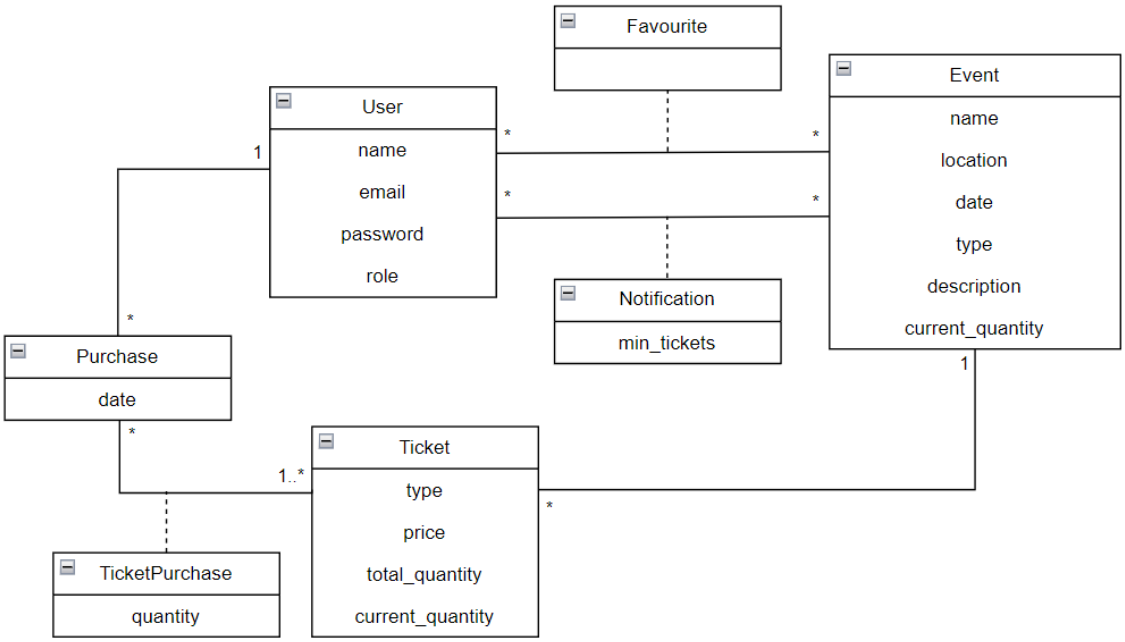

Fig. 1. UML with the model of TickETCD.

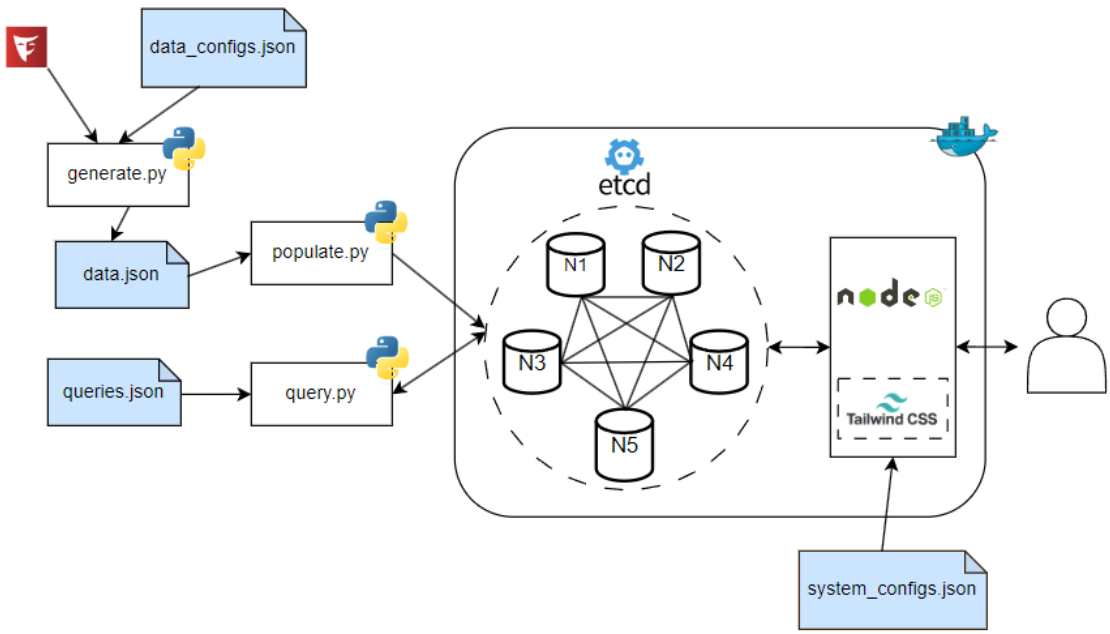

Fig. 2. Architecture Diagram representing the structure of TickETCD.

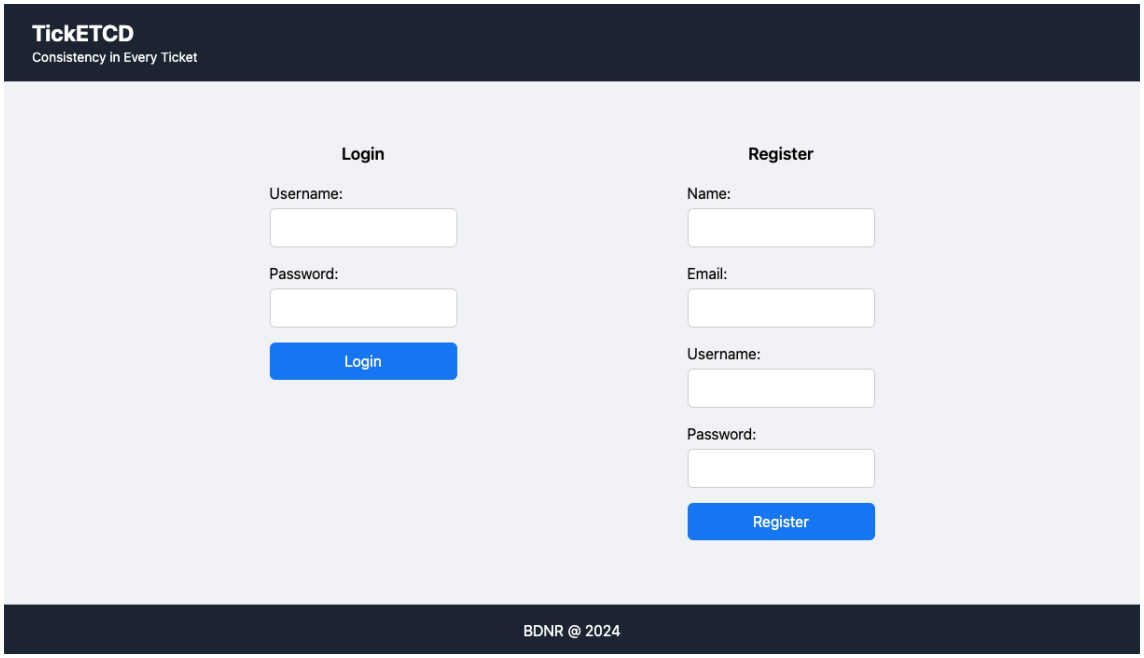

Fig. 3. Login/register page, where a user can register if not have a account yet or login in their account.

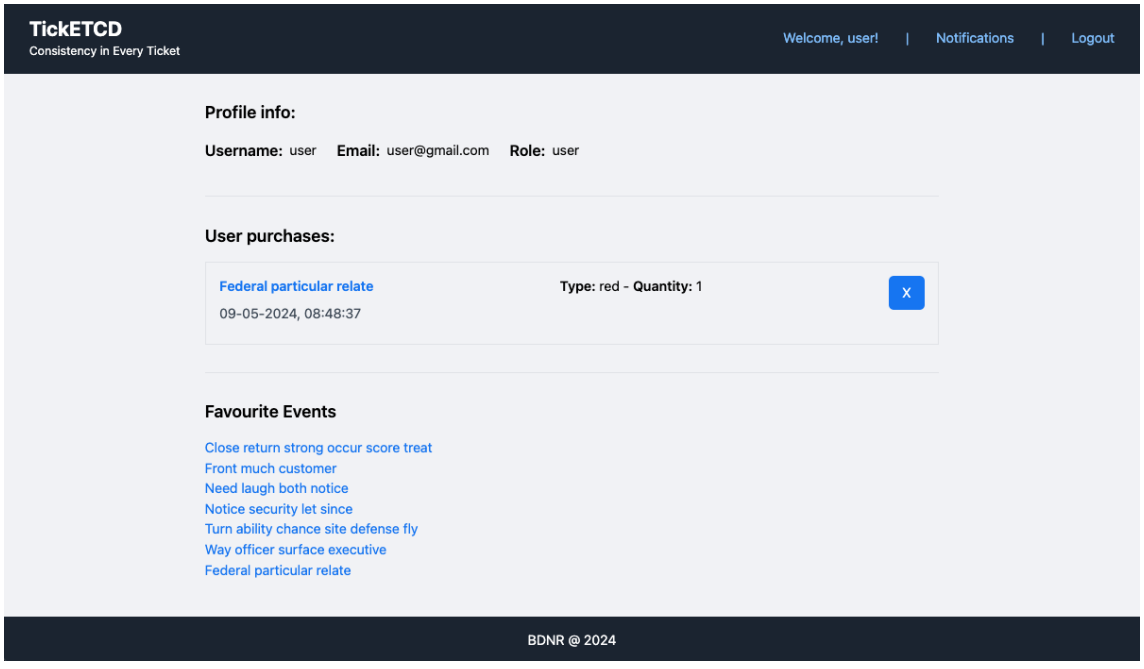

Fig. 4. Profile page, where user can see the information about themselves, the events that they marked as favourite, the historic about the purchases done and cancel them.

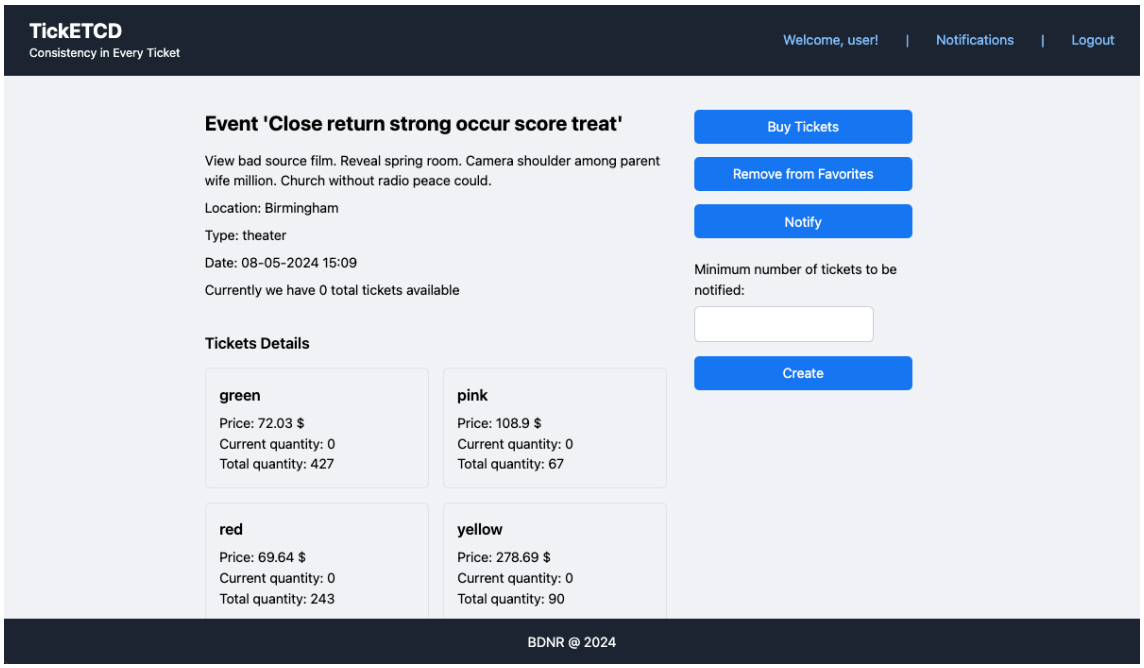

Fig. 5. Event page, where a user can see the details about a certain event, buy tickets, add that event to favourite or be notify when that event has less than x tickets.

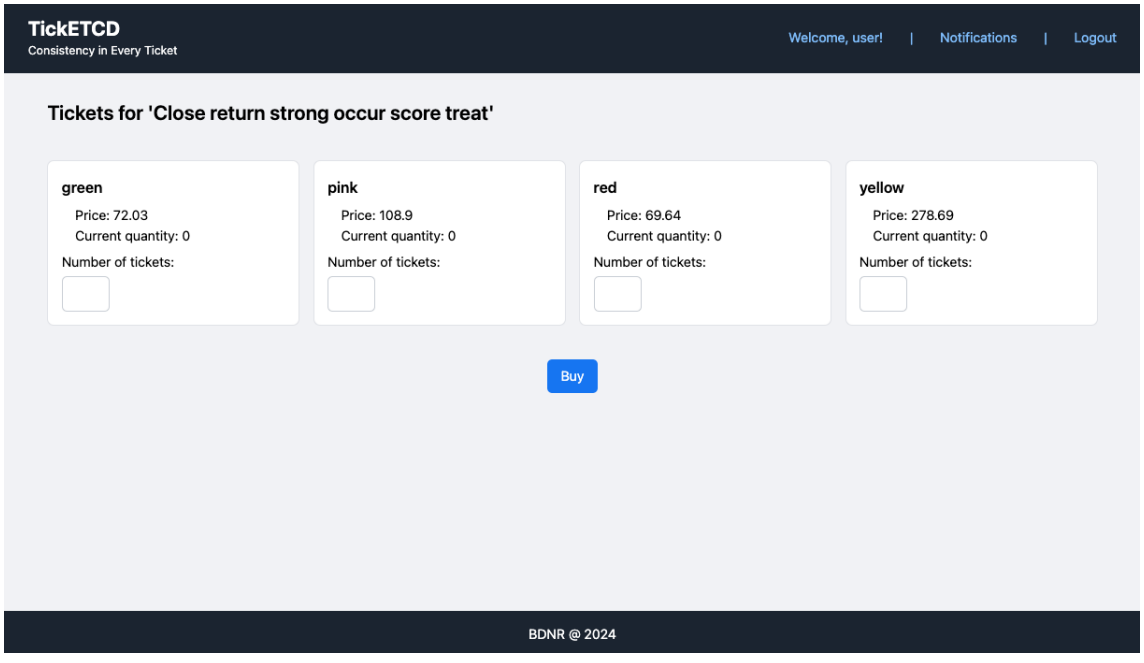

Fig. 6. Purchase page, where the user can select the number of tickets they want to buy of each type.

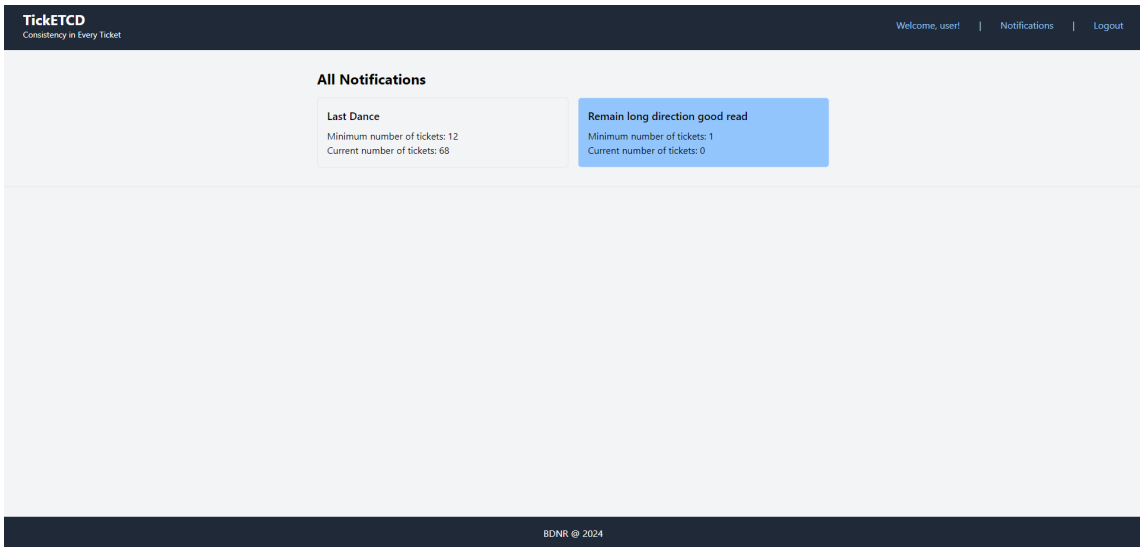

Fig. 7. Notifications page, where user can see all the notifications that they created, the ones that are active and the ones that are not.

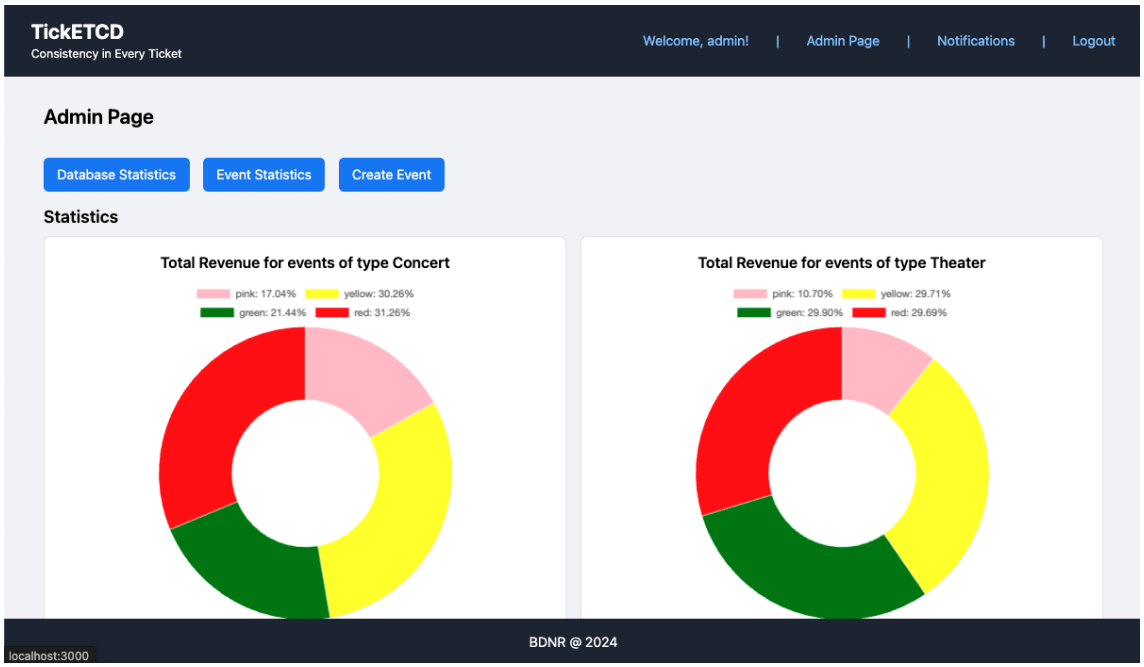

Fig. 8. Admin page showing the statistics about the events in the platform.

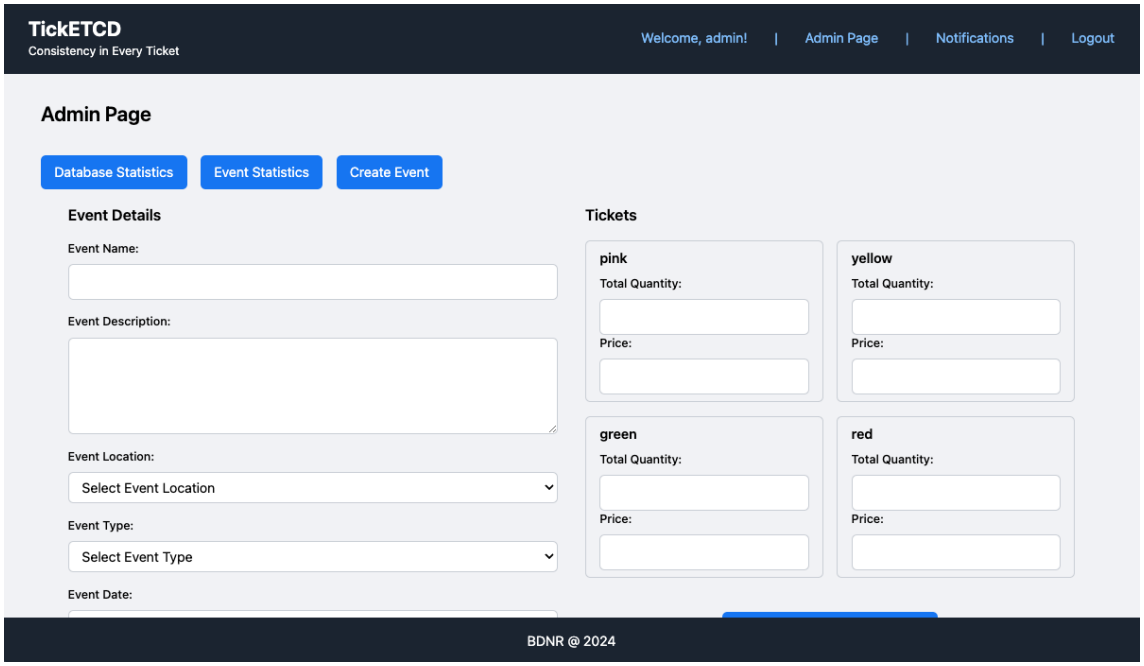

Fig. 9. Admin page for creation of a new event in the platform.

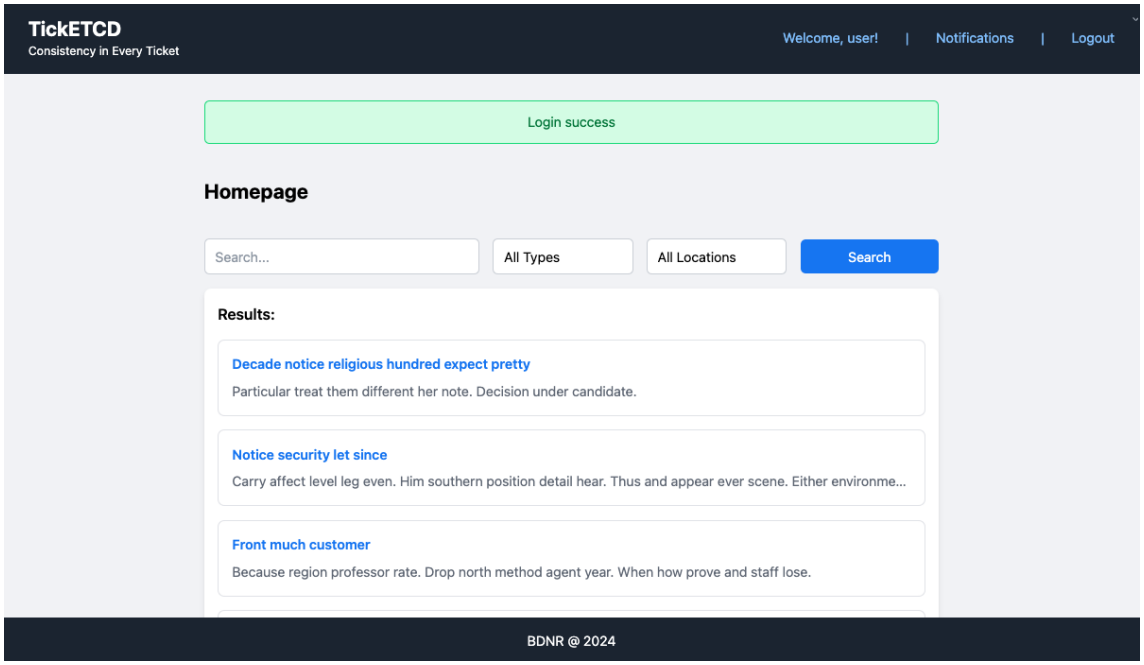

Fig. 10. Home page, where a user can search for a certain event.

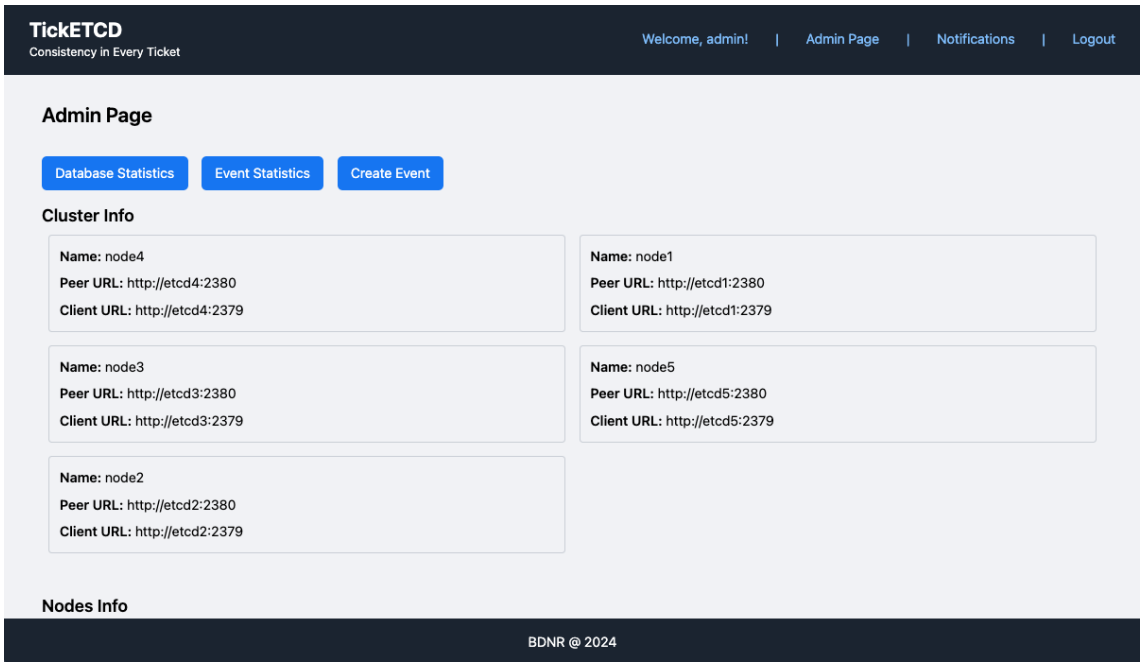

Fig. 11. Admin page showing information related to the nodes of the cluster.

REFERENCES

[1] Creative Commons Attribution 3.0. 2024. *Raft Consensus Algorithm.* Retrieved May 03, 2024 from https://raft.github.io/
[2] Aerospike. 2024. *Key-value databases.* Retrieved April 19, 2024 from https://aerospike.com/what-is-a-key-value-store/
[3] CloudFlare. 2023. *HTTP and TLS.* Retrieved May 07, 2024 from https://www.cloudflare.com/en-gb/learning/ssl/transport-layer-security-tls/
[4] CodeAcademy. 2023. *CRUD.* Retrieved May 08, 2024 from https://www.codecademy.com/article/what-is-crud
[5] Consul Community. 2023. *Consul.* Retrieved April 29, 2024 from https://www.consul.io
[6] CoreOS. 2021. *Clustering Guide.* Retrieved April 30, 2024 from https://etcd.io/docs/v3.4/op-guide/clustering/
[7] CoreOS. 2021. *Data Model.* Retrieved May 03, 2024 from https://etcd.io/docs/v3.3/learning/data_model/
[8] CoreOS. 2021. *Frequently Asked Questions.* Retrieved May 06, 2024 from https://etcd.io/docs/v3.3/faq/
[9] CoreOS. 2021. *Hardware recommendations.* Retrieved May 10, 2024 from https://etcd.io/docs/v3.3/op-guide/hardware/
[10] CoreOS. 2021. *How to watch keys.* Retrieved May 08, 2024 from https://etcd.io/docs/v3.5/tutorials/how-to-watch-keys/
[11] CoreOS. 2021. *Interacting with etcd.* Retrieved May 02, 2024 from https://etcd.io/docs/v3.6/dev-guide/interacting_v3/
[12] CoreOS. 2021. *KV API guarantees.* Retrieved May 09, 2024 from https://etcd.io/docs/v3.3/learning/api_guarantees/
[13] CoreOS. 2021. *System limits.* Retrieved May 05, 2024 from https://etcd.io/docs/v3.4/dev-guide/limit/
[14] CoreOS. 2021. *Wht etcd.* Retrieved May 05, 2024 from https://etcd.io/docs/v3.1/learning/why/
[15] CoreOS. 2022. *etcd API.* https://etcd.io/docs/v3.5/learning/api/#transaction
[16] CoreOS. 2023. *etcd Community.* Retrieved April 21, 2024 from https://etcd.io/community/
[17] CoreOS. 2023. *etcd Github.* Retrieved April 28, 2024 from https://github.com/etcd-io/etcd/discussions
[18] CoreOS. 2023. *etcd Google groups.* Retrieved May 08, 2024 from https://groups.google.com/g/etcd-dev
[19] CoreOS. 2023. *etcd Twitter.* Retrieved May 03, 2024 from https://twitter.com/etcdio
[20] CoreOS. 2024. *CoreOS.* Retrieved April 10, 2024 from https://fedoraproject.org/coreos/
[21] CoreOS. 2024. *etcd.* Retrieved April 30, 2024 from https://etcd.io/
[22] CoreOS. 2024. *ETCD CLI.* Retrieved May 09, 2024 from https://github.com/etcd-io/etcd/tree/main/etcdctl
[23] CoreOS. 2024. *etcd documentation.* Retrieved May 01, 2024 from https://etcd.io/docs/v3.5/
[24] CoreOS. 2024. *ETCD Libraries.* Retrieved May 01, 2024 from https://etcd.io/docs/v3.3/integrations/
[25] DB-engines. 2024. *DB-Engines Ranking of Key-value Stores.* Retrieved May 09, 2024 from https://db-engines.com/en/ranking/key-value+store
[26] Docker. 2023. *Docker documentation.* Retrieved May 09, 2024 from https://www.docker.com
[27] Apache Software Foundation. 2023. *Apache ZooKeeper.* Retrieved April 21, 2024 from https://zookeeper.apache.org
[28] Extio Foundation. 2023. *Deep Dive into etcd: A Distributed Key-Value Store.* Retrieved May 01, 2024 from https://medium.com/@extio/deep-dive-into-etcd-a-distributed-key-value-store-a6a7699d3abc
[29] GeeksforGeeks. 2023. *etcd ACID properties.* Retrieved May 07, 2024 from https://www.geeksforgeeks.org/acid-properties-in-dbms/
[30] GeeksforGeeks. 2023. *Introduction of B-Tree.* Retrieved May 09, 2024 from https://www.geeksforgeeks.org/introduction-of-b-tree-2/
[31] Red Hat. 2021. *What was CoreOS and CoreOS container Linux?* Retrieved May 12, 2024 from https://www.redhat.com/en/technologies/cloud-computing/openshift/what-was-coreos
[32] IBM. 2024. *What is etcd?* Retrieved April 27, 2024 from https://www.ibm.com/topics/etcd
[33] IBM. 2024. *What is the CAP Theorem?* Retrieved May 08, 2024 from https://www.ibm.com/topics/cap-theorem
[34] Microsoft. 2023. *etcd.* Retrieved May 03, 2024 from https://github.com/microsoft/etcd3/tree/master/src/test
[35] Microsoft. 2023. *ETCD3.* Retrieved May 02, 2024 from https://github.com/microsoft/etcd3
[36] Microsoft. 2024. *Impedance Mismatch.* Retrieved May 06, 2024 from https://devblogs.microsoft.com/oldnewthing/20180123-00/?p=97865
[37] MongoDB. 2024. *NoSQL.* Retrieved Mach 02, 2024 from https://www.mongodb.com/resources/basics/databases/nosql-explained
[38] NodeJS. 2023. *NodeJS.* Retrieved May 10, 2024 from https://nodejs.org/en/download
[39] Database of databases. 2022. *Database of databases.* Retrieved May 09, 2024 from https://dbdb.io/db/etcd
[40] Mathew Palmer. 2024. *How Does Kubernetes Use etcd?* Retrieved May 12, 2024 from https://matthewpalmer.net/kubernetes-app-developer/articles/how-does-kubernetes-use-etcd.html
[41] Python. 2023. *Python Faker.* Retrieved April 12, 2024 from https://faker.readthedocs.io/en/master/
[42] RedHat. 2024. *RedHat.* Retrieved May 07, 2024 from https://www.redhat.com/en
[43] Redis. 2023. *Redis Database.* Retrieved May 05, 2024 from https://redis.io
[44] TailwindCSS. 2023. *TailwindCSS.* Retrieved May 10, 2024 from https://tailwindcss.com
[45] Louis Taylor. 2022. *Python-etcd.* Retrieved May 03, 2024 from https://github.com/kragniz/python-etcd3