Temporal GraphQL: A Tree Grammar Approach

Curtis E. Dyreson
Department of Computer Science, Utah State University, Logan, UT, USA

Bishal Sarkar
Department of Computer Science, Utah State University, Logan, UT, USA

Abstract

This paper presents a novel system, called Temporal GraphQL, for supporting temporal data in web services. A temporal web service is a service that provides a *temporal* view of data, that is, a view of the current data as well as past or future states of the data. Capturing the history of the data is important in data forensics, data auditing, and subscriptions, where an application continuously reads data. GraphQL is a technology for improving the development and management of web services. Originally developed by Facebook and widely used in industry, GraphQL is a query language for web services. This paper introduces Temporal GraphQL. We show how to use tree grammars to model GraphQL schemas, data, and queries, and propose temporal tree grammars to model Temporal GraphQL. We extend GraphQL with temporal snapshot, slice, and delta operators. To the best of our knowledge, this is the first work on Temporal GraphQL and temporal tree grammars.

2012 ACM Subject Classification Information systems → Temporal data; Information systems → Service discovery and interfaces; Information systems → Query languages

Keywords and phrases Temporal databases, temporal queries, GraphQL, web services

Digital Object Identifier 10.4230/LIPIcs.TIME.2025.9

1 Introduction

Web applications rely on *web services* for data management. A web service is a an *application programming interface* (API) endpoint that allows a client to interact with a back-end database over the web. Web services are ubiquitous; when on-line users shop, make dinner reservations, buy airline tickets, vote, or post social media updates each interaction typically invokes several web services. Web services read and write data formatted in Javascript Objection Notation (JSON). JSON is a lightweight, text notation for representing objects. Though JSON DBMSs are rising in popularity, e.g., MongoDB ranks fifth in a recent ranking of DBMS popularity [15], JSON is the most widely used data exchange language. JSON is tightly integrated into many modern programming languages, e.g., Python, Java, Typescript, all have libraries to quickly convert objects formatted in JSON to objects in the host language and vice-versa.

GraphQL is a technology for improving the development and management of web services [22]. Originally developed by Facebook and widely used in industry, GraphQL is a query language for a web service, or more generally, an API. GraphQL supports *queries* that read data as well as *mutations* that update data on the types provided by the schema. A GraphQL query is evaluated to produce JSON data requested by a user.

GraphQL, however, lacks support for temporal data. Temporal data is data annotated with time metadata. This paper presents a novel system, called Temporal GraphQL, for supporting temporal data in web services. A temporal web service is a service that provides a *temporal* view of data, that is, a view of the current data as well as past or future states of the data. Capturing the history of the data is important in data forensics, data auditing, and subscriptions, where an application continuously reads data. For a subscription, instead of returning all of the data in each snapshot, only the differences between snapshots can be provided. This "delta" is usually much smaller than the entire dataset.

This paper makes the following contributions.

- We show how to use tree grammars to model GraphQL schemas, data, and queries, and propose temporal tree grammars to model temporal GraphQL. To the best of our knowledge, this is the first work on temporal tree grammars.
- We extend GraphQL with temporal snapshot, slice, and delta operators.

This paper is organized as follows. The next section reviews GraphQL. Section 3 introduces Temporal GraphQL. We then describe tree grammars and how they are used in supporting Temporal GraphQL in Section 4. The final two sections cover related work and conclusions.

2 Review of GraphQL

In this section we review GraphQL. The starting point for GraphQL is a *schema*, which describes the types provided by an API as described in Section 2.1. GraphQL supports queries and mutations (updates) using the API. For our purposes mutations are a variation on queries, so this paper focuses exclusively on queries, which are presented in Section 2.2.

2.1 GraphQL Schemas

A GraphQL schema describes the types provided by an API. As an example, consider the GraphQL schema specification shown in Figure 1. The schema is taken from GraphQL's tutorial for learning GraphQL [23] and has three object *types*: `Character`, `Planet`, and `Species`. The miniworld for the types is a science fiction world where a character originates on some planet, is of some species, and has friends who are characters. Each type has one or more properties. A property represents a key-value pair in JSON. The name of the property is the key and the schema records type constraints on the value. The `Character` type has `name`, `friends`, `homeworld`, and `species` properties. The `Character name` is a `String` type and must be non-null (indicated by the "`!`"). The `friends` property is a list (indicated by the enclosing brackets "`[]`") of references to `Characters`.

The data is checked during query evaluation to ensure that it conforms to the schema, if not, an error is generated. As an example Figure 3 shows a fragment of a data instance that conforms to the schema given in Figure 1. Each JSON object implicitly contains an `id` property that uniquely identifies the object within the data collection, not just the type (collection-wide unique identifiers are used to implement client-side caching of objects). Sub-objects are represented as references, e.g., the `homeworld` property in the `Character` type is a reference to a `Planet` object.

2.2 Queries

Queries are at the core of GraphQL. Every GraphQL schema must also support an entry point for queries, this type is known as the `Query` type. The `Query` type specifies the root entry point(s) for the database. An example is given in Figure 2. The `Query` type has three entry points:

1. `hero` reads a `Character`,
2. `characters` yields a list `Characters`, and
3. `planets`, which returns a list of `Planets`.

The entry point is the starting point for the query, more fields can be added to flesh out objects and sub-objects in a query result. An example is given in Figure 4. In the example, the `hero` entry point is expanded to include the `name` and `friends` properties. The `friends`

```
type Character {
  name: String!
  friends: [Character]
  homeworld: Planet
  species: Species
}

type Planet {
  name: String
  climate: String
}

type Species {
  name: String
  lifespan: Int
  origin: Planet
}
```

Figure 1 A GraphQL schema for a science fiction database.

```
type Query {
  hero: Character
  characters: [Character]
  planets: [Planet]
}
```

Figure 2 The `Query` type in GraphQL defines query “entry points”.

```
Character = [
  { "id": "id_r2_d2",
    "name": "R2-D2",
    "friends": [
      "characterId": "id_luke_skywalker",
      "characterId": "id_han_solo"
    ],
    "homeworldId": "id_naboo",
    "species": "id_droid" },
   { "id": "id_luke_skywalker",
     "name": "Luke Skywalker",
     ... },
   ...
]
Planet = [
  { "id": "id_naboo",
    "name": "Naboo",
    "climate": "Temperate" },
  ...
]
Species = [
  { "id": "id_droid",
     "name": "Droid",
    ... }
  ...
]
```

Figure 3 JSON that conforms to the GraphQL schema in Figure 1 for a science fiction database.

property is a list of `Characters`, and for each sub-object in the list the `name` and `homeworld` (which is a `Planet` type object) properties are selected. An example of the result of evaluating the query is given in Figure 5.

Queries can also include *filters*, which are predicates to test the data for membership in the result. For example, suppose that we want to select the character named Luke Skywalker. Then we could filter the `hero` entry point in a query as follows:

```
hero(filter: { name: { eq: "Luke Skywalker" } }) {
    name
    ...
}
```

Entry points can also be specified to take arguments. For instance, we could modify the `hero` entry point to match a specific name as follows.

```
type Query {
    hero(name: String) : Character
    ...
}
```

```
query {
  hero {
    name
    friends {
      name
      homeworld {
        name
        climate
      }
    }
  }
}
```

Figure 4 A query using the `hero` entry point.

```
{ "data": {
    "hero": {
      "name": "R2-D2",
      "friends": [
        {  "name": "Luke Skywalker",
            "homeworld": {
              "name": "Tatooine",
              "climate": "Desert"
            }
        },
        ....
      ]
    }
}
```

Figure 5 A fragment of the result of the GraphQL query in Figure 4.

The query to fetch Luke Skywalker would then be as follows.

```
hero(name: "Luke Skywalker") {
    name
    ...
}
```

3 Temporal GraphQL

Science fiction data changes over time as edits to the data are made and new data is inserted. This section describes how to capture the the changing history. In Temporal GraphQL the data is assumed to be annotated with time metadata that records the lifetime of the data in some temporal dimension. Common dimensions are transaction time and valid time. We first show how to add support for time to a schema type, we then describe several temporal query operators that let users travel in time and select past versions of data. In the next section we discuss how to support Temporal GraphQL in a layered approach where a temporal schema/operator is translated into the corresponding GraphQL schema/operator. This implementation strategy leverages non-temporal GraphQL to support GraphQL.

3.1 Temporal Types

In a temporal GraphQL schema a type can be made temporal by adding a GraphQL *directive* as shown in Figure 6. A GraphQL directive is prefixed with the "`@`" character. Directives are essentially decorators in many popular programming languages. Directives can be added to both schemas and queries. The `@temporal` directive indicates that the type is now a temporal type, that is the schema type will represent data annotated with temporal metadata. For the purposes of this paper we assume a single, transaction-time temporal dimension as it is common for systems to record the time data is created and deleted. Extensions to valid time, belief time, or bitemporal times are future work. We will further assume that the transaction time is recorded as a period or interval timestamp, rather than a temporal element, that

```
# @temporal directive declaration
directive @temporal() on OBJECT | SCHEMA

# Character is a temporal type
type Character @temporal {
  name: String
  friends: [Character]
  homeworld: Planet
  species: Species
}
...
```

Figure 6 A GraphQL schema for a temporal science fiction database.

```
query @slice({start: 3, stop: 4}) ( {
  hero {
    name
  }
}
```

Figure 7 An example `@slice` query.

is as a set of periods, or as an indeterminate time [2, 19]. Extending to handle temporal elements or temporal indeterminacy are left to future work. The schema can be a mix of temporal and nontemporal types, though for this running example we will assume that all of the object types have been similarly annotated.

3.2 Temporal Queries

We assume that an API or web service supplies the temporal data in a temporal GraphQL instance. Temporal GraphQL supports several kinds of *temporal queries*. Each kind of query is specified by a GraphQL *query directive* that modifies the behavior of the query as described below.

- `directive @snapshot(time: Int!) on QUERY` - A snapshot query takes as input a time, t, and returns the non-temporal data as of t.
- `directive @slice(time: Timestamp!) on QUERY` - A slice query takes a `Timestamp` object and returns the temporal history as of the given timestamp. The `Timestamp` type is defined in Figure 9.
- `directive @current() on QUERY` - The current query returns the snapshot as of the current time.
- `directive @delta(time: Timestamp!) on QUERY` - The delta query takes a `Timestamp` and returns all of the data that changed during the `Timestamp`.

Figure 7 shows how a slice query would be specified.

4 A Layered Approach to Supporting Temporal GraphQL

In this section we describe a layered approach to mapping a temporal schema (query) into a *representational* schema (query). The key to the approach is to model the schema as a *temporal tree grammar*. Section 4.1 provides background on tree grammars, which are extended in Section 4.2 to include support for time. The temporal tree grammar models the representational schema and data as described in Section 4.2.1.

4.1 A Tree Grammar Approach to Modeling GraphQL Schemas

Introduced in 1969, a tree grammar is a (context-free) grammar for generating (or parsing) trees [33].

▶ **Definition 1** (Simple Unordered Tree Grammar). *A simple unordered tree grammar is a four-tuple* (Σ, R, N, Δ) *where:*

- Σ *is the alphabet, a finite set of terminals;*
- N *is a finite set of nonterminals;*
- $R \in N$ *is a set of root (start) nonterminals, and*
- Δ *is a finite set of productions, with the following properties. Each production is of the form* $n \to x_1[\alpha_1]\beta_1 \ \dots \ x_k[\alpha_k]\beta_k$ *where* $n \in N$*, the body of the production is unordered (the production represents all possible permutations of the body of the production), and for all* i*,*
  - $x_i \in \Sigma$*;*
  - α_i *is a well-formed formula of terminals or nonterminals, square brackets to indicate tree nesting, and metalanguage symbols from the EBNF (e.g.,* $*$ *represents Kleene closure);*
  - *and* β_i *is the empty string, ? to indicate 0 or 1 occurrences, or* $*$ *for Kleene closure.*

The grammar is for *unranked* trees [4]. In a *ranked* tree grammar each terminal or nonterminal would have a fixed arity or "rank", but GraphQL trees (and JSON) are best modeled by unranked tree grammars, so for instance a `homeworld` node may have between one and two descendants since `name` and `climate` are optional. The grammar is also context-free and simple because every clause in the body starts with a terminal (the root of a subtree) so a parser or generator for the grammar can deterministically choose the clause in the body of the production based on a single lookahead token (the evaluation only has to keep track of which clause was chosen).

One use of the tree grammar is to ensure that a GraphQL query is *valid.* Validation is the first step in query evaluation. Validation consists of checking the query against the schema to determine if all of the fields correspond to a type property and that the types are nested correctly. It is straightforward to convert a GraphQL schema to a tree grammar. The conversion introduces one production for every type (including the `Query` type). For example the converted tree grammar for the GraphQL schema of Figure 1 is given below. Note that a query is agnostic about JSON lists in the result so this distinction is not made in the grammar to validate a query.

- $\Sigma = \{$`id`, `name`, `friends`, `homeworld`, `species`, `climate`, `lifespan`, `origin`, `characters`, `hero`, `planets`$\}$
- $R = Q$
- $N = \{Q, C, P, S\}$
- Δ contains the following productions (for each $x[\alpha]\beta$ in the body β is a "?" because the clause is optional, so for clarity we will omit the "?").
  $Q \to$ `hero`$[C]$ `characters`$[S]$ `planets`$[P]$
  $C \to$ `id`$[\,]$ `name`$[\,]$ `friends`$[C]$ `homeworld`$[P]$ `species`$[S]$
  $P \to$ `id`$[\,]$ `name`$[\,]$ `climate`$[\,]$
  $S \to$ `id`$[\,]$ `name`$[\,]$ `lifespan`$[\,]$ `origin`$[P]$

The grammar can be used to determine whether a query, such as that given in Figure 4 is valid. Figure 8 shows the tree representation of the query in Figure 4. The grammar is used to validate from the root down by first choosing to expand Q (the start nonterminal) by `hero`$[C]$ based on the root of the query matching `hero`. C is then expanded by `name`$[\,]$ and `friends`$[C]$, and so on.

A GraphQL schema can also be converted to a *result* grammar that describes the result of a query. Below is the converted result grammar. Note that lists in the result are represented in a tree as repeated children (using Kleene closure in the grammar) and that *null* values could be present.

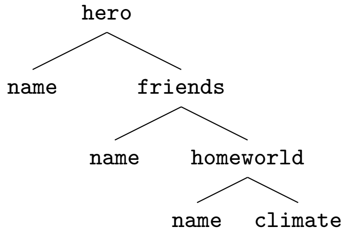

Figure 8 The tree corresponding to the query in Figure 4.

- $\Sigma = \{$`id`, `name`, `friends`, `homeworld`, `species`, `climate`, `lifespan`, `origin`, `String`, `data`, `ID`, $null\}$
- $R = D$
- $N = \{D, Q, C, P, S\}$
- Δ contains the following productions (for each $x_i[\alpha_i]\beta_i$ in the body there is a "?" because the clause is optional, so for clarity we will omit the "?").
  $D \rightarrow \texttt{data}[Q]$
  $Q \rightarrow [\ C \mid S* \mid P*\ ]$
  $C \rightarrow \texttt{id}[\texttt{ID}]\ \ \texttt{name}[\ \texttt{String} \mid null\ ]\ \ \texttt{friends}[C]*\ \ \texttt{homeworld}[P]\ \ \texttt{species}[S]$
  $P \rightarrow \texttt{id}[\texttt{ID}]\ \ \texttt{name}[\ \texttt{String} \mid null\ ]\ \ \texttt{climate}[\ \texttt{String} \mid null\ ]$
  $S \rightarrow \texttt{id}[\texttt{ID}]\ \ \texttt{name}[\ \texttt{String} \mid null\ ]\ \ \texttt{lifespan}[\ \texttt{String} \mid null\ ]\ \ \texttt{origin}[P]$

The result grammar is used to generate the result shown in Figure 5.

4.2 Temporal Tree Grammar

To support temporal data, we introduce a *temporal tree grammar*. There are (at least) two ways in which a grammar can be temporal.

1. The grammar itself evolves - The grammar changes over time as rules are updated, inserted, and deleted. For Temporal GraphQL this is akin to schema evolution or versioning [32].
2. The data changes over time - The grammar describes a snapshot of the data, but the data itself is temporal, capturing the entire timeline of its evolution (in some time dimension(s)). The non-temporal grammar parses each snapshot rather than the entire history of the data.

We consider evolution of the grammar next, and in Section 4.2.2 representing the data's history.

4.2.1 Grammar Evolution

To record the changing history of the schema, each terminal and nonterminal in the body of a production is annotated with a transaction time lifetime (transaction time since edits to the schema are transactions). The lifetimes are updated as follows when edits are made to the productions (to the schema).

- Deletion of terminal or nonterminal in the body - When part of a production is deleted the lifetime of the deleted parts is ended at the time at which it was deleted.
- Insertion of a terminal or nonterminal - An insertion creates a new lifetime starting at the time at which the part of the production was inserted and terminating at time *uc* (until changed).

- Insertion or deletion of a production - Each terminal or nonterminal in the body is updated as either inserted (lifetime starts) or deleted (lifetime ends).
- Terminal or nonterminal marked as *deprecated* - GraphQL supports a `@deprecated` annotation in the schema. Rather than deleting properties or types, they can be annotated as deprecated. Since deprecated properties/types are not deleted, their lifetime will continue (since they contine to be present in the schema). Since deprecated properties and types are part of the GraphQL standard, Temporal GraphQL will continue to support such types, so a slice at the current time will include deprecated fields present as of the time of the slice. However, in Temporal GraphQL fields and properties (including deprecated fields) can be (logically) deleted since rollback to previous versions of the schema is supported. So in some sense, Temporal GraphQL "fixes" the need for having a `@deprecated` annotation, but for compatibility, it continues to support the annotation.
- Change to a production - We model the change to a production as the deletion of the changed part, followed by an insertion of the new part.

For example, suppose that at time 1 the temporal type given in Figure 6 is inserted in the schema. Then at time 2 the `name` field is deleted and at time 3 a `moniker` field is added. By time 4 the production for the type in the query grammar would be as shown below, the temporal annotations are shown as subscripts for each terminal and nonterminal in the body of the production.

$$C \rightarrow \texttt{id}_{1-uc}[\,]\ \texttt{name}_{1-2}[\,]\ \texttt{moniker}_{3-uc}[\,]\ \texttt{friends}_{1-uc}[C]\ \texttt{homeworld}_{1-uc}[P]\ \texttt{species}_{1-uc}[S]$$

The temporal annotations specify the time(s) at which the body of the production is valid. For instance the rule would be used as follows in validating a query, such as that given in Figure 4. Initially, the lifetime of the query is 1-*uc*. But the query contains the `Character name` property. So as the query `name` construct is parsed, the lifetime becomes 1-*uc* $\bigcap$ 1-2 = 1-2, which is the time interval in which the `name` property existed. If the lifetime becomes empty, then the parsing fails (at no transaction time is the query valid).

The lifetime computed by the query grammar can be propagated to the task of generating the result using the result grammar, in particular it constrains the result to only data that was alive during the computed lifetime as described next

4.2.2 A Representational Model for Data Evolution

A result tree grammar can be converted to a temporal result tree grammar that includes timestamps in the data and supports multiple versions of a property as follows. First, for each nonterminal, X, in the grammar corresponding to an `@temporal` object type we add add two nonterminals: a *version* nonterminal, represented as X_V, which represents a single version of the data, and a *history* nonterminal, represented as X_T, which is a list of versions. Next, we add a production, $X_T \rightarrow \texttt{versions}X[X_V*]$, to indicate that a history nonterminal is a list of versions of type X. We also add a production, $X_V \rightarrow \texttt{timestamp}[T]\ \texttt{data}[X]$, to state that a version is the paring of a timestamp (represented by `Timestamp` type, T) and the data for that version, which is an instance of X. Finally, in the body of each production we replace X with X_T to indicate that X is temporal.

As an example, the productions in the temporal result tree grammar is given below.

- $\Sigma = \{\texttt{id}, \texttt{name}, \texttt{friends}, \ldots \texttt{lifespan}, \texttt{origin}, \texttt{String}, \texttt{data}, \texttt{ID}, \mathit{null}\}$
- $R = Q$
- $N = \{Q, C, C_T, C_V, P, P_T, P_V, S, S_T, S_V\}$
- Δ contains the following productions.

$D \rightarrow$ `data`$[Q]$
$Q \rightarrow [\ C_T \mid S_T* \mid P_T*\ ]$
$C_T \rightarrow$ `versionsCharacter`$[C_V*]$
$C_V \rightarrow$ `timestamp`$[T]$ `data`$[C]$
$C \rightarrow$ `id[ID]` `name`$[$ `String` $\mid$ *null* $]$ `friends`$[C]*$ `homeworld`$[P]$ `species`$[S]$
$P_T \rightarrow$ `versionsPlanet`$[P_V*]$
$P_V \rightarrow$ `timestamp`$[T]$ `data`$[P]$
$P \rightarrow$ `id[ID]` `name`$[$ `String` $\mid$ *null* $]$ `climate`$[$ `String` $\mid$ *null* $]$
$S_T \rightarrow$ `versionsSpecies`$[S_V*]$
$S_V \rightarrow$ `timestamp`$[T]$ `data`$[S]$
$S \rightarrow$ `id[ID]` `name`$[$ `String` $\mid$ *null* $]$ `lifespan`$[$ `String` $\mid$ *null* $]$ `origin`$[P]$
$T \rightarrow$ `start`$[$ `Int` $\mid$ *null* $]$ `stop`$[$ `Int` $\mid$ *null* $]$

A key feature of the result tree grammar is that it can be expressed with a non-temporal GraphQL schema, enabling GraphQL itself to support Temporal GrapQL. The *representational schema* for the temporal result grammar is given n Figure 9. The representational schema is a GraphQL schema that represents the data and temporal metadata. In the representational schema the `Character` type is converted to a `CharacterTemporal` type. The `CharacterTemporal` type replaces `Character` everywhere else in the schema. A temporal type is a list of *versions*. Each version is a `Timestamp` paired with a snapshot of a non-temporal `Character` object. The `Timestamp` object is simply a transaction time interval, but in practice could be bitemporal, a temporal element, or indeterminate; the object is defined to represent the kinds and nature of times that annotate the data.

The temporal result grammar describes the types produced by the API. A fragment of the representational data is shown in Figure 10. The data represents one change to the "`R2-D2`" `Character`, the `homeworld` was updated from "`Tatooine`" to "`Naboo`". The change creates a new version of the `Character` object.

We note that this representational grammar takes a *temporal-centric* approach to querying (described in the next section). In a temporal centric approach filtering is done primarily by temporal constraints rather than value-specific constraints. There are alternative, potential representations that we leave to future work that could better support a *value-centric* approach. For instance, a `Timestamp` property could be added to each type in the schema to record its lifetime. We envision a system in which a designer could choose the representational schema that best suits their needs or alternatively choose support for more than one kind of representational schema simultaneously. Note that the schema is not used for storage of the data, rather it provides a view for query access, so supporting alternative representations should be possible.

4.2.3 Evaluating Temporal Queries

The temporal result grammar is also used to construct the result of temporal queries. Just as in the non-temporal case, the grammar is used to *generate* a result by repeated application of the productions starting from the start nonterminal. But in the temporal case two additional features are needed, the timestamps have to be processed and the result can be temporal or non-temporal (depending on the operation). Let's consider the `@slice` operator first.

We introduce the notion of *sequenced generation* (or parsing) to process `@slice`. It is sequenced because for each use of a production in the generation (parse) there is an associated *lifetime* that represents when the data is alive. The lifetime is important to track since in the generated tree a child cannot exist without its parent, so each child's lifetime is constrained

```
...

type CharacterTemporal {
  versionsCharacter: [CharacterVersion]
}

type CharacterVersion {
  timestamp: Timestamp
  snapshot: Character
}

type Character {
  name: String
  friends: [CharacterTemporal]
  homeworld: PlanetTemporal
  species: SpeciesTemporal
}

...

type Timestamp {
  start: String
  stop: String
}
```

Figure 9 A representational GraphQL schema for the temporal schema of Figure 1.

```
...
CharacterTemporal = [
  { "id": "id_r2_d2",
    "versionsCharacter": ["id_r2_d2_v1",
                          "id_r2_d2_v2"] },
  ...
]
CharacterVersion = [
  { "id": "id_r2_d2_v1",
    "timestamp": "id_t1",
    "snapshot": "id_r2_d2_s1 },
  { "id": "id_r2_d2_v2",
    "timestamp": "id_t2",
    "snapshot": "id_r2_d2_s2 },
]
Character = [
  { "id": "id_r2_d2_s1",
    "name": "R2-D2",
    "friends": [
      "characterId": "id_luke_skywalker_v1",
      "characterId": "id_han_solo_v1"  ],
    "homeworldId": "id_tatooine_v1",
    "species": "id_droid_v1" },
  { "id": "id_r2_d2_s2",
    "name": "R2-D2",
    "friends": [
      "characterId": "id_luke_skywalker_v1",
      "characterId": "id_han_solo_v1" ],
    "homeworldId": "id_naboo_v1",
    "species": "id_droid_v1" },
   ...
]
...
```

Figure 10 JSON that conforms to the GraphQL schema in Figure 1 for a science fiction database.

by the lifetime of its parent. A slice grabs the part of a history within a time interval specified by the user. So initially, the lifetime of the root is given in the slice. Moreover, when the `@slice` is validated using an evolving grammar, the validation produces a timestamp that represents the times at which the query is valid (matches the temporal schema grammar). So the lifetime is the intersection of the time specified in the slice and the time produced by the validation. If this time is empty, then the `@slice` generates an error (the schema does not match the query). As the generation proceeds from the root to each leaf in the result, the lifetime is maintained along each branch in the tree by taking the intersection of the branch's lifetime with the data's lifetime in any version object.

As an example, consider the `@slice` query given in Figure 7 using the grammar of Section 4.2 including the change made to the grammar in Section 4.2.1 on the data of Figure 10. The query specifies a slice from 3-4 but validation produces a time of 1-2 since `name` only exists at time 1-2. The intersection is empty, hence an error would be generated. But suppose the slice was from 1-1. The result depicted in Figure 11 would be generated. Initially 1-1 would be passed from the root (the `hero` node) along each branch

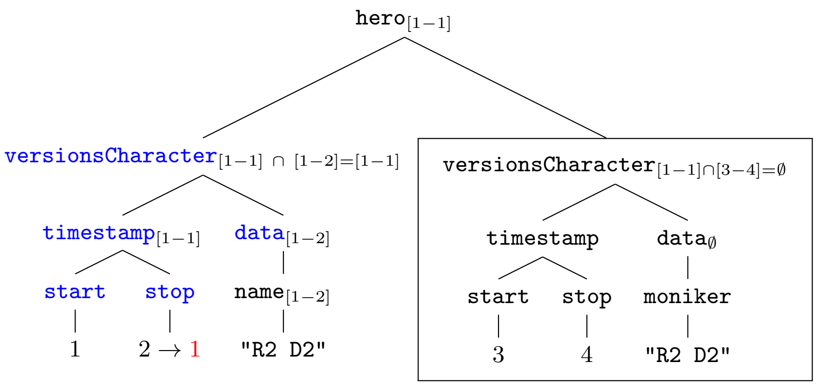

Figure 11 The tree corresponding to the query in Figure 4.

(each `Character` version) in the constructed tree. For each version in the data the intersection of the version's time and the branch's time is computed, which becomes the branch's time for descendants along the branch, and the timestamp is updated. If the time is empty then the branch generation is terminated. As an example, for the first (leftmost) `versionsCharacter` node in Figure 11 the `stop` property in the timestamp is changed from 2 to 1 as indicated in a red font. For the second `versionsCharacter` branch the intersection of 1-1 and 3-4 (the time of the version) is empty, so the framed branch in the tree is not generated in the result.

The evaluation of `@delta` starts the same as the evaluation of `@slice` with the initial computation of the lifetime of the root. But the intersection of times along a branch is not performed, instead branches are pruned if the version lifetime does not start or end during the branch's lifetime. So if we replace `@slice` with `@delta` in the query in Figure 7 and use a `start` time of 1, then the validation would produce a root timestamp of times 1-2. When applied to Figure 11 the leftmost branch (without changing the timestamp) would be selected, but the right child has a time of 3-4, so is outside the interval 1-2 and would not be included in the result.

Finally, the evaluation of `@snapshot` (and `@current`) is similar to that of `@slice` insofar as branches are pruned that fall outside of the slice time. But all non-data nodes are removed from the result. As example using Figure 11 the `@snapshot` at time 1 would prune the framed subtree rooted at the rightmost `versionsCharacter` node (since its lifetime is 3-4), and also prune from the result the non-data nodes by placing the `name` node as the only child of the `hero` node (removing the nodes in blue font in the leftmost `versionsCharacter` subtree).

5 Related Work

To the best of our knowledge there have been no previous papers on Temporal GraphQL or temporal tree grammars. There has, however, been extensive previous research to supporting temporal data [3, 20, 30]. This research has fallen into two broad categories: versioning and timestamp-based support. Timestamp-based queries are common in temporal relational databases. A temporal relational database [25] stores data that is annotated with time metadata. The time metadata records when the data was alive in some time domain, e.g., transaction time [27], valid time [28], or both. Such databases can be queried in various ways. For instance in TSQL2 [34] a query can be evaluated to retrieve the data's history e.g., a timeslice query [26], or retrieve the data as of some time instant, e.g., a snapshot query [35], or perform a query at every time instant in the data's history, e.g., a sequenced query [5]. But TSQL2 does not support queries that ask for versions of data, e.g., get the second version of an employment record or retrieve the changes to the employment record. Data versioning is more common in temporal object-oriented databases [13] or temporal documents where each edit or change creates a new version of an object or document [21]. Users can navigate among the versions and restore old versions if necessary.

Semi-structured data representations such as JSON, XML, and YAML are used to represent both data and documents and thus need to support both timestamp and version histories [1, 7–11, 16, 31, 37]. Semi-structured data changes over time, sometimes frequently, as new data is inserted and existing data is edited and deleted [12, 24, 29]. Previous research in temporal XML and JSON called elements that maintain their identity over time *items* [14, 17, 18, 36]. Items are timestamped with a lifetime and as an element can be moved within a document. Each change to an item creates a version, which is also timestamped. Previous research showed how to represent, query, describe with a schema and validate temporal semi-structured data. Differences in XML and JSON spawned further research in schema validation and versioning for JSON data [6].

6 Conclusions

GraphQL is a widely used technology for making it easier to develop and maintain web applications. GraphQL queries and mutations are used to read and write data to a back-end database through a web services API. But data and schemas change over time and capturing and querying this history is important in many applications. In this paper we presented Temporal GraphQL, a technology that adds support for time to GraphQL. We observed that tree grammars can model GraphQL schemas, data, and queries and we proposed temporal tree grammars to model temporal GraphQL. And we extended GraphQL with temporal snapshot, slice, and delta operators. The key advantage of our design is that it leverages non-temporal GraphQL to support GraphQL.

Our short-term future work is targeted to implementing Temporal GraphQL as a layer for a GraphQL system. But such a system has to be coupled with techniques for converting web services to temporal web services, to supply the data for the temporal types in the query, which in term needs temporal support in a back-end database. We are currently working on a PostGraphile layer; PostGraphile is a GraphQL system for Postgres databases. We also plan to specialize the `@temporal` annotation to support different kinds of time, e.g., `@validTime`. And we plan to add temporal elements and support for indeterminacy to the `Timestamp` type. Indeterminacy will also require some changes to queries to support indeterminate queries. Finally, we plan to generalize the support for temporal metadata outlined in this paper to include other kinds of metadata, such as quality metadata.

References

References

1 Toshiyuki Amagasa, Masatoshi Yoshikawa, and Shunsuke Uemura. A Data Model for Temporal XML Documents. In *Database and Expert Systems Applications, 11th International Conference, DEXA 2000, London, UK, September 4-8, 2000, Proceedings*, pages 334–344, 2000. doi:10.1007/3-540-44469-6_31.

2 Luca Anselma, Luca Piovesan, and Paolo Terenziani. Dealing with temporal indeterminacy in relational databases: An AI methodology. *AI Commun.*, 32(3):207–221, 2019. doi:10.3233/AIC-190619.

3 Alessandro Artale, Roman Kontchakov, Alisa Kovtunova, Vladislav Ryzhikov, Frank Wolter, and Michael Zakharyaschev. Ontology-mediated query answering over temporal data: A survey (invited talk). In *24th International Symposium on Temporal Representation and Reasoning, TIME 2017, October 16-18, 2017, Mons, Belgium*, pages 1:1–1:37, 2017. `doi:10.4230/LIPIcs.TIME.2017.1`.

4 Michael Benedikt, Leonid Libkin, and Frank Neven. Logical definability and query languages over ranked and unranked trees. *ACM Trans. Comput. Logic*, 8(2):11–es, April 2007. `doi:10.1145/1227839.1227843`.

5 Michael H. Böhlen and Christian S. Jensen. Sequenced semantics. In *Encyclopedia of Database Systems, Second Edition*. Springer, 2018. `doi:10.1007/978-1-4614-8265-9_1053`.

6 Safa Brahmia, Zouhaier Brahmia, Fabio Grandi, and Rafik Bouaziz. τjschema: A framework for managing temporal json-based nosql databases. In *Database and Expert Systems Applications - 27th International Conference, DEXA 2016, Porto, Portugal, September 5-8, 2016, Proceedings, Part II*, pages 167–181, 2016. `doi:10.1007/978-3-319-44406-2_13`.

7 Zouhaier Brahmia, Fabio Grandi, Safa Brahmia, and Rafik Bouaziz. A graphical conceptual model for conventional and time-varying json data. *Procedia Computer Science*, 184:823–828, 2021. The 12th International Conference on Ambient Systems, Networks and Technologies (ANT) / The 4th International Conference on Emerging Data and Industry 4.0 (EDI40) / Affiliated Workshops. `doi:10.1016/j.procs.2021.03.102`.

8 Zouhaier Brahmia, Fabio Grandi, Safa Brahmia, and Rafik Bouaziz. τjupdate: An update language for time-varying JSON data. *J. Comput. Lang.*, 79:101258, 2024. `doi:10.1016/J.COLA.2024.101258`.

9 Zouhaier Brahmia, Hind Hamrouni, and Rafik Bouaziz. XML data manipulation in conventional and temporal XML databases: A survey. *Comput. Sci. Rev.*, 36:100231, 2020. `doi:10.1016/J.COSREV.2020.100231`.

10 Sudarshan S. Chawathe, Serge Abiteboul, and Jennifer Widom. Representing and Querying Changes in Semistructured Data. In *Proceedings of the Fourteenth International Conference on Data Engineering, Orlando, Florida, USA, February 23-27, 1998*, pages 4–13, 1998. `doi:10.1109/ICDE.1998.655752`.

11 Shu-Yao Chien, Vassilis J. Tsotras, and Carlo Zaniolo. Efficient Schemes for Managing Multiversion XML Documents. *VLDB J.*, 11(4):332–353, 2002. `doi:10.1007/s00778-002-0079-4`.

12 Junghoo Cho and Hector Garcia-Molina. The Evolution of the Web and Implications for an Incremental Crawler. In *VLDB*, pages 200–209, 2000. URL: `http://www.vldb.org/conf/2000/P200.pdf`.

13 Carlo Combi. Temporal object-oriented databases. In *Encyclopedia of Database Systems, Second Edition*. Springer, 2018. `doi:10.1007/978-1-4614-8265-9_404`.

14 Faiz Currim, Sabah Currim, Curtis E. Dyreson, Richard T. Snodgrass, Stephen W. Thomas, and Rui Zhang. Adding Temporal Constraints to XML Schema. *IEEE Trans. Knowl. Data Eng.*, 24(8):1361–1377, 2012. `doi:10.1109/TKDE.2011.74`.

15 DB-Engines Ranking. `https://db-engines.com/en/ranking`. Accessed: 2025-05-10.

16 Curtis E. Dyreson and Fabio Grandi. Temporal XML. In *Encyclopedia of Database Systems, Second Edition*. Springer, 2018. `doi:10.1007/978-1-4614-8265-9_411`.

17 Curtis E. Dyreson and Kalyan G. Mekala. Prefix-Based Node Numbering for Temporal XML. In *Web Information System Engineering - WISE 2011 - 12th International Conference, Sydney, Australia, October 13-14, 2011. Proceedings*, pages 172–184, 2011. `doi:10.1007/978-3-642-24434-6_13`.

18 Curtis E. Dyreson, Amani M. Shatnawi, Sourav S. Bhowmick, and Vishal Sharma. Temporal JSON keyword search. *Proc. ACM Manag. Data*, 2(3):177, 2024. `doi:10.1145/3654980`.

19 Curtis E. Dyreson and Richard Thomas Snodgrass. Supporting valid-time indeterminacy. *ACM Trans. Database Syst.*, 23(1):1–57, March 1998. `doi:10.1145/288086.288087`.

20 Opher Etzion, Sushil Jajodia, and Suryanarayana M. Sripada, editors. *Temporal Databases: Research and Practice. (the book grow out of a Dagstuhl Seminar, June 23-27, 1997)*, volume 1399 of *Lecture Notes in Computer Science*. Springer, 1998. doi:10.1007/BFb0053695.

21 Aayush Goyal and Curtis E. Dyreson. Temporal JSON. In *5th IEEE International Conference on Collaboration and Internet Computing, CIC 2019, Los Angeles, CA, USA, December 12-14, 2019*, pages 135–144. IEEE, 2019. doi:10.1109/CIC48465.2019.00025.

22 GraphQL: A Query Language for Your API. https://graphql.org. Accessed: 2025-05-10.

23 Introduction to GraphQL. https://graphql.org/learn. Accessed: 2025-05-10.

24 Panagiotis G. Ipeirotis, Alexandros Ntoulas, Junghoo Cho, and Luis Gravano. Modeling and Managing Content Changes in Text Databases. In *ICDE*, pages 606–617, 2005. doi:10.1109/ICDE.2005.91.

25 Christian S. Jensen and Richard T. Snodgrass. Temporal database. In *Encyclopedia of Database Systems, Second Edition*. Springer, 2018. doi:10.1007/978-1-4614-8265-9_395.

26 Christian S. Jensen and Richard T. Snodgrass. Timeslice operator. In *Encyclopedia of Database Systems, Second Edition*. Springer, 2018. doi:10.1007/978-1-4614-8265-9_1426.

27 Christian S. Jensen and Richard T. Snodgrass. Transaction time. In *Encyclopedia of Database Systems, Second Edition*. Springer, 2018. doi:10.1007/978-1-4614-8265-9_1064.

28 Christian S. Jensen and Richard T. Snodgrass. Valid time. In *Encyclopedia of Database Systems, Second Edition*. Springer, 2018. doi:10.1007/978-1-4614-8265-9_1066.

29 Venkata N. Padmanabhan and Lili Qiu. The Content and Access Dynamics of a Busy Web Site: Findings and Implications. In *SIGCOMM*, pages 111–123, 2000. doi:10.1145/347059.347413.

30 Vangipuram Radhakrishna, P. V. Kumar, and V. Janaki. A survey on temporal databases and data mining. In *Proceedings of the The International Conference on Engineering & MIS 2015*, ICEMIS '15, pages 52:1–52:6, New York, NY, USA, 2015. ACM. doi:10.1145/2832987.2833064.

31 Flavio Rizzolo and Alejandro A. Vaisman. Temporal XML: Modeling, Indexing, and Query Processing. *VLDB J.*, 17(5):1179–1212, 2008. doi:10.1007/s00778-007-0058-x.

32 John F. Roddick. A survey of schema versioning issues for database systems. *Inf. Softw. Technol.*, 37(7):383–393, 1995. doi:10.1016/0950-5849(95)91494-K.

33 William C. Rounds. Context-free grammars on trees. In *Proceedings of the 1st Annual ACM Symposium on Theory of Computing, May 5-7, 1969, Marina del Rey, CA, USA*, pages 143–148. ACM, 1969. doi:10.1145/800169.805428.

34 Richard T. Snodgrass, editor. *The TSQL2 Temporal Query Language*. Kluwer, 1995.

35 Richard T. Snodgrass, Michael H. Böhlen, Christian S. Jensen, and Andreas Steiner. Transitioning temporal support in TSQL2 to SQL3. In *Temporal Databases: Research and Practice. (the book grow out of a Dagstuhl Seminar, June 23-27, 1997)*, pages 150–194, 1997. doi:10.1007/BFb0053702.

36 Richard T. Snodgrass, Curtis E. Dyreson, Faiz Currim, Sabah Currim, and Shailesh Joshi. Validating Quicksand: Temporal Schema Versioning in tauXSchema. *Data Knowl. Eng.*, 65(2):223–242, 2008. doi:10.1016/j.datak.2007.09.003.

37 Fusheng Wang and Carlo Zaniolo. An XML-Based Approach to Publishing and Querying the History of Databases. *World Wide Web*, 8(3):233–259, 2005. doi:10.1007/s11280-005-1317-7.