Compound Schema Registry (Extended Abstract)

Silvery Fu and Xuewei Chen
UC Berkeley

Abstract

Schema evolution is the process of modifying a database system's schema to maintain compatibility with existing data [1–3, 6]. It allows data producers to update schemas while ensuring they remain compatible with the ones used by downstream consumers. For example, a producer might add a new `timestamp` field that does not disrupt existing consumers unprepared for this change.

A schema registry [4, 9] is a common approach aiming to address the challenges of schema evolution, especially for real-time data streaming. It serves as a centralized repository to store, manage, validate, and ensure the compatibility of schemas. The registry facilitates communication between producers and consumers through a well-defined data contract encapsulated within a schema. It controls schema evolution through clear and explicit compatibility rules, ensuring that all participants adhere to established standards. The registry optimizes data transmission by using schema IDs instead of full schema definitions. At runtime, the schema registry dynamically resolves these IDs to their corresponding schemas, enabling systems to correctly interpret incoming data streams and integrate schema changes without interruptions.

However, existing schema registries can typically manage only simple modifications to schemas, such as adding or removing fields. More complex **syntactic** alterations, such as renaming fields, changing data types, or modifying units and scaling, are generally considered breaking changes. These changes can lead to application downtime, requiring a human in the loop to write schema matching and mapping code at the application level to restore compatibility and carefully manage the migration. For instance, in a Kafka ecosystem that includes a data consumer, producer, and schema registry [4], developers responsible for the consumer application must be notified to update their code before the producer makes any changes to field names or types. Such coordination is crucial to ensure that the consumer continues to receive data correctly. This process can be tedious and often prevents scenarios such as zero-downtime upgrades; it also limits the ability of applications to access real-time data from data sources with previously unknown or changing schemas.

To this end, we propose *generalizing* automatic schema evolution to accommodate a broader range of schema syntax changes. With generalized schema evolution (GSE), as long as the **semantics** of two fields or schemas remain equivalent or compatible—as determined by the data consumer—data streams will continue uninterrupted when the data producer evolves the schema. We argue that to realize GSE, the schema registry should transform into a compound AI system [11]. Our insight is that Large Language Models (LLMs), with their capability to understand data semantics, can significantly improve how schema changes are managed and streamline the *schema mapping* between different schema versions. For example, consider two versions of motion sensor schemas illustrated in Fig. 2. Our approach would enable the automatic mapping of data from version v2 to version v1, allowing data consumers operating under the v1 schema to continue accessing data produced under v2.

We present a design and a prototype for *compound schema registry* to support GSE, which aims to address three key requirements: **(1) Accurate:** The mappings across schema versions must be precise, ensuring correct generation and application of transformations to fields and values within the schema to the data records. **(2) Fast and efficient:** Rather than using LLMs to directly translate each data record—a process that can be inefficient and slow due to frequent model calls—we should generate schema mappings with LLMs and translate them into dataflow operations implemented on the data path (*e.g.,* at the data consumer, within the message broker, or integrated into the data pipeline). This approach of *generating off-path code for on-path execution* not only ensures high accuracy but also improves efficiency. **(3) Transparent:** The mapping process and its outputs should be straightforward and easily verifiable for correctness, avoiding opaque operations (*e.g.,* hidden within a single model call). Moreover, we advocate for creating an *intermediate representation* for schema mappings that are independent of specific dataflow languages. This approach simplifies the generation process by avoiding any unnecessary intricacies of individual language syntax, while enabling the reuse of mappings across different data processing engines and platforms.

```yaml
kind: "Motion sensor"
name: "v1"
description: "Philips Hue"
fields:
  - name: "motion"
    type: "boolean"
    description: >
      True if motion
      is detected.
    required: true

  - name: "enabled"
    type: "boolean"
    description: >
      True when the sensor
      is activated, false
      when deactivated.
    required: true

  - name: "sensitivity"
    type: "integer"
    description: >
      Motion sensitivity
    default: 2
    min: 0
    max: 4
```

```yaml
kind: "Motion sensor"
version: "v2"
description: "Vivint"
fields:
  - name: "triggered"
    type: "boolean"
    description: >
      Indicates whether the
      sensor has been
      triggered.
    required: true

  - name: "enabled"
    type: "boolean"
    description: >
      Indicates whether the
      motion sensor is enabled
      (True) or bypassed (False).
    required: true

  - name: "battery_percentage"
    type: "integer"
    description: >
      Measures the current battery
      level of the motion sensor.
    required: true
```

Figure 1: Example schemas for motion sensor data (v1 and v2).

To meet these requirements, instead of directly generating the dataflow operators from the given source and target schemas, we propose a **task-specific language** for schema mapping called Schema Transformation Language (STL). The language defines a collection of schema mapping commands, as detailed in Table 1. These include (i) *schema matching* commands for assessing compatibility between schemas, (ii) *field transformation* commands for directly modifying schema fields such as adding, deleting, or renaming them, and (iii) *value transformation* commands for converting field values to comply with new schema specifications. Each command handles a specific sub-task of schema mapping. At runtime, the schema registry uses STL as part of the prompt to invoke an LLM, where each command is defined as a function, *e.g.,* using the OpenAI function calling interface in our prototype, along with the two versions of the schemas to be mapped. The LLM then generates schema mappings as STL commands, as depicted in Fig. 2. Subsequently, an *assembler* translates these STL commands into the corresponding dataflow operations using the dataflow language of the target platform, which can then be patched or installed on the data pipeline.

Command class	Command name	Description
Schema matching	MATCH	Used to determine whether the source and target schemas correspond to the same entity; if they match, the schema mapping will continue; otherwise, it will abort.
Field transformation	COPY	Directly copies data from the source field to the target field without any transformation.
	ADD	Inserts a new field into the target schema that does not exist in the source schema.
	CAST	Converts the data type of the source field to match the expected type of the target field.
	DELETE	Removes the field from the source schema when it is not required in the target schema.
	RENAME	Changes the name of the source field to match the name of the target schema.
	DEFAULT	Assigns a predefined default value to a target field when data is unavailable or null.
	MISSING	Used when no appropriate mapping exists to map the source field to a target field, implying a schema mapping failure.
Value transformation	SCALE	Adjusts the numerical values in the source field by a specified factor for the target field.
	SHIFT	Modifies the values in the source field by adding or subtracting a constant value.
	LINK	Establishes a correspondence between values in the source field and defined values in the target field, used for fields with enum type.
	GEN	Generate a transformation function that defines how to convert values from the source field to fit the target field's requirements.
	APPLY	Applies a transformation function, either generated or predefined by the developer, to the value of a source field to derive the value of the target field.

Table 1: Key commands in Schema Transformation Language (STL) of the compound schema registry.

```
{from: triggered, to: motion, transformation: RENAME triggered TO motion}
{from: battery_percentage, to: None, transformation: DELETE battery_percentage}
{from: None, to: sensitivity, transformation: ADD sensitivity TYPE integer}
{from: sensitivity, to: sensitivity, transformation: DEFAULT sensitivity TO 2}
{from: enabled, to: enabled, transformation: COPY}
```

Figure 2: Generated mappings for motion sensor schema v1 and v2.

Source schema	Target schema	Precision		Recall		F1	
		STL	Base	STL	Base	STL	Base
Philips Hue	Vivint	**0.91**	0.73	**0.98**	0.83	**0.94**	0.78
SimpliSafe	Vivint	**1**	0.2	**0.8**	0.2	**0.89**	0.2
SimpliSafe	Philips Hue	**1**	0.8	**0.9**	0.67	**0.95**	0.72

Table 2: Accuracy of evolving schema with STL and direct model call.

Our initial results suggest promising improvements in schema mapping accuracy with STL compared to generating dataflow operators directly using an LLM. For example, when applied to real-world IoT device schemas and schema evolution scenarios, the STL approach can significantly improve the average F1 score—measured based on the precision and recall of generating the correct mappings—from 78% to 94% across runs for the example schemas, as shown in Table 2. This is because STL: (1) breaks down the schema mapping task into smaller, specific sub-tasks (*e.g.,* field transformations to value transformations for each field), and (2) separates mapping generation from dataflow generation so that each step can be performed more easily. With better per-STL-command prompt engineering, this approach could achieve even higher mapping accuracy. Further, we found that the quality of schema definitions (*e.g.,* how concise each field explanation is) plays an important role in mapping accuracy. We assume the schema definitions are given or can be extracted automatically in a separate process [10], which itself can also be performed through a compound AI approach. An interesting question is how we can co-design the schema extraction, mapping, and evolution processes.

We are extending the prototype to handle schema evolution across different target platforms and evaluating it using various datasets [7, 8], while comparing it with prior approaches [5, 6, 12]. Our codebase will be made available at https://llmint.org.

Design Pattern: Task-Specific Language / IR. We propose extending the discussed design pattern beyond schema evolution by employing LLMs to generate messages in a task-specific language for broader applications. Within this framework, each command is clearly defined to handle a specific sub-task, with predefined templates for inputs and outputs. Such unambiguous and modular task specification can also make the output verifiable and task execution debuggable. This approach can deliver more *general and reliable* LLM-based automation across various domains, such as workflow automation, data automation, and decision support systems.

References

[1] 2024. Diving Into Delta Lake: Schema Enforcement & Evolution. https://www.databricks.com/blog/2019/09/24/diving-into-delta-lake-schema-enforcement-evolution.html.

[2] 2024. Hudi Schema Evolution. https://hudi.apache.org/docs/schema_evolution/.

[3] 2024. Schema Evolution in Confluent. https://developer.confluent.io/patterns/event-stream/schema-evolution.

[4] 2024. Schema Registry. https://docs.confluent.io/platform/current/schema-registry/index.html.

[5] Zui Chen et al. 2023. Seed: Domain-specific data curation with large language models. *arXiv e-prints* (2023), arXiv–2310.

[6] Carlo Curino, Hyun Jin Moon, Alin Deutsch, and Carlo Zaniolo. 2013. Automating the database schema evolution process. *The VLDB Journal* 22 (2013).

[7] Michael De Jong, Arie van Deursen, and Anthony Cleve. 2017. Zero-downtime SQL database schema evolution for continuous deployment. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. IEEE, 143–152.

[8] Mark Lukas Möller, Meike Klettke, and Uta Störl. 2020. EvoBench–a framework for benchmarking schema evolution in NoSQL. In *2020 IEEE International Conference on Big Data (Big Data)*. IEEE, 1974–1984.

[9] Rahul Sharma, Mohammad Atyab, Rahul Sharma, and Mohammad Atyab. 2022. Schema Registry. *Cloud-Native Microservices with Apache Pulsar: Build Distributed Messaging Microservices* (2022), 81–101.

[10] Michael Stonebraker et al. 2013. Data curation at scale: the data tamer system.. In *CIDR*, Vol. 2013.

[11] Matei Zaharia et al. 2024. The Shift from Models to Compound AI Systems. https://bair.berkeley.edu/blog/2024/02/18/compound-ai-systems/.

[12] Yunjia Zhang et al. 2023. Schema matching using pre-trained language models. In *Proc. IEEE ICDE*.