NORTHWEST NAZARENE UNIVERSITY

Using Python and MySQL to audit records provided in JSON format

THESIS

Submitted to the Department of Mathematics and Computer Science in partial fulfillment of the requirements for the degree of BACHELOR OF SCIENCE

Timothy D. Zink

2024

THESIS

Submitted to the Department of Mathematics and Computer Science in partial fulfillment of the requirements for the degree of BACHELOR OF SCIENCE

Timothy D. Zink

2024

Using Python and MySQL to audit records provided in JSON format

Author: Timothy Zink

Timothy Zink

Approved: Barry Myers

Barry L. Myers, Ph.D., Department of Mathematics and Computer Science, Faculty Advisor

Approved: BJ Howard

BJ Howard, Programmer/Analyst, Department of Administrative Software, Second Reader

Approved: Dale A Hamilton

Dale Hamilton, Ph.D., Chair, Department of Mathematics & Computer Science

Abstract

Using Python and MySQL to audit records provided in JSON format.

ZINK, TIMOTHY (Department of Mathematics and Computer Science), MYERS, DR. BARRY (Department of Mathematics and Computer Science), HAMILTON, DR. DALE (Department of Mathematics and Computer Science).

An audit database stores a history of records from another database, accompanied by other helpful information, such as who changed the record and when. An audit is useful when issues occur, such as when a record is changed or deleted when it should not be. A history or "audit" of records provides valuable information for those investigating and can even be used to restore a record to a previous state. As the university transitions to a new enterprise resource planning (ERP) system that lacks sufficient auditing, a need arises for a way to audit a table of records represented in JSON format. This project developed a proof of concept for a utility (or module) that handles the logic behind auditing records and interacts with an audit database. Programming this module started with the initial specifications, which were refined as further requirements became known. While the project delivered a working module, there are areas where the design could be improved.

Acknowledgments

I want to thank my family for their continuous support throughout my education. Additionally, A huge thanks to the university's IT department for their support and cooperation, with special thanks to the help desk staff and TAs. I greatly appreciate their assistance in editing this thesis and preparing for the Senior Seminar.

Table of Contents

List of Figures

Key Terms

Application Programming Interface (API) – A set of procedures that provides access to data and features for external systems.

Audit – A history of a record. It also refers to adding a snapshot to the existing history.

Database Triggers – SQL commands that are run automatically when specific actions occur on a database.

Enterprise Resource Planning (ERP) – A system that manages data and processes for a business.

JavaScript Object Notation (JSON) – A text format representing various data. It is often used to serialize data communicated between two programs/systems that do not necessarily use the same language.

Library – A collection of Python modules that can be downloaded or installed for use in Python scripts.

Log File – A file used to record information such as warnings and errors.

Module – A Python script file that can be imported into other Python scripts and provides additional functionality.

MySQL – One of the many relational database management systems.

Primary Key – A column on a database table that uniquely identifies records in the table.

Python – A programming language used to write scripts.

Script – A file containing code.

Serialize/Deserialize – The process of converting data to a different format (such as JSON).

Snapshot – The process of determining and recording changes that have occurred to records.

Structure Query Language (SQL) – A language used in relational database systems.

Introduction

Overview

As the university transitions to a new Enterprise Resource Planning (ERP) system that lacks sufficient auditing, there is a need for an alternative auditing method. An audit snapshot of a collection of records in JSON format that is periodically retrieved can be used instead to track changes. These changes can then be stored in an external database to maintain a history. This project developed a code module that accepts a collection of records, audits them, and stores the results in a database table.

Background

An audit stores a history of records accompanied by other helpful information, such as who changed the record and when. For this thesis, a "record" or "original record" refers to the actual record. An "audit record" (shown in Figure 1) refers to a copy of the original record with some "audit metadata" added to it, stored in an "audit table" in an "audit database." Audit metadata refers to additional data about changes to the record, such as when the change was made. A single original record may have many audit records, each representing the original record at a single point in time, effectively creating a history of the original record. All the audit records for each original record in a table are stored in the same audit table; the history of a table is stored in its corresponding audit table. It is common to store all the audit tables in a separate database (called an "audit database") to further protect the audit's integrity. These audits provide helpful information when investigating inconsistencies and allow a previous record version to be restored.

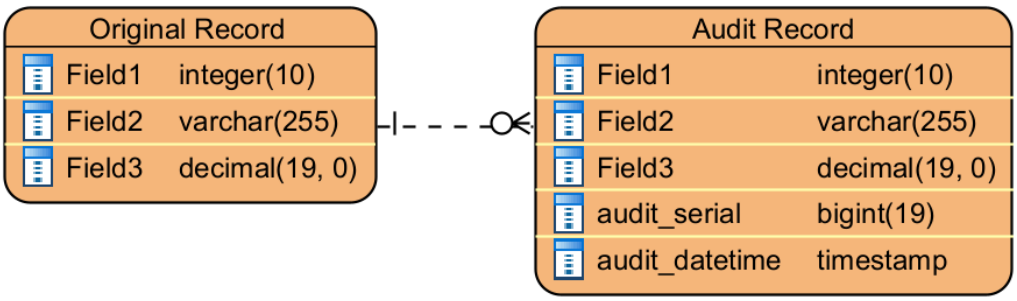


Figure 1. Example record and its audit record.

Currently, the university uses an Enterprise Resource Planning (ERP) system that allows direct access and modifications to its internal database. Auditing can be done easily using database triggers to automatically audit records as they are modified.

However, the university is transitioning to a new ERP system that does not permit direct access or modification to its internal database but does have an Application Programming Interface (API) that outputs records in JSON format (see Figure 2, Appendix A). This project's goal was to construct a code module that can audit a collection of records in JSON format and store the audit in an external database (or "audit database"). This project does not include interacting with the new ERP system; it only addresses interacting with the audit database and handling the logic for detecting changes.

```
[
   {
      "firstName": "Timothy",
      "lastLast": "Zink",
      "age": 21,
      "height_meters": 1.9,
      "isFake": false,
   },
   {
      "firstName": "John",
      "lastName": "Doe",
      "age": null,
      "heightMeters": 1.8,
      "isFake": true
   }
]
```

Figure 2. Example collection of records in JSON format.

Specifications

The project specifications and requirements were determined during discussions with the project supervisor. Below are the initial specifications when the project was started. Some additional requirements were defined as situations arose. Requirements that were added or changed during the project are addressed in the Development section.

The deliverable for this project was a program module or library (called a "code module") that fulfills the objective below. The module could be used directly by another program or script, or the module could be used as a reference for developing a new solution in the future.

Objective

Create a code module that can be imported into a script and creates a "snapshot" of a given collection of records. The snapshot tracks changes to the records (creation, deletion, and updates) between two points in time.

Requirements

I. The project's resulting code module needs to be able to handle JSON format.

II. Save each snapshot in a database. The audit records should include the date and time of when the snapshot was taken.

III. The code module must be reliable. It must behave consistently and gracefully handle unexpected or unlikely situations.

IV. The module must log any errors in a log file (not just print them to the screen) to assist in troubleshooting problems.

Constraints

I. The database used will be a MySQL database.

II. The language used to write the code module may only be Python or Perl.

III. The code module needs to be Linux-compatible. In general, avoid using any features dependent on a specific operating system.

Development

The initial specifications only determined some characteristics for the development of the project. Decisions were made as the development progressed, and additional requirements were added. Following is a discussion of the decisions, justifications, and alternatives.

Decisions

The specifications permitted the programming language to be either Python or Perl. Although many of the scripts already used by the university are written in Perl, Python was chosen as the language for the code module (Python Software Foundation, 2001-2024). It is a higher-level language that is more feature rich than Perl and has many libraries available.

While the type of database for storing the audit was already determined to be MySQL (Oracle, 2024), the project was left to decide how to interact with the database. One of the options was to use a library, such as SQLAlchemy (SQLAlchemy, 2007-2024), to interact with the audit database. Such a library would simplify much of the code and effectively hide the complexity of working with a database. While abstraction like this would be advantageous in

most situations, one of the additional requirements was that the structure of the audit table needs to be updated when the structure of the records changes (see item I in the Additional Requirements Summary section). This required more control over the database than what was provided by the libraries investigated. After a discussion with the project supervisor, the decision was made to interact with the database directly using SQL commands.

To meet the requirement of logging errors, a custom "logger" was implemented for the code module to use. The logging requirements were expanded to include any SQL commands sent to the database and any other information that might be useful for fixing issues with the code module if any occurred (see item II in the Additional Requirements Summary section). Additional logging statements were added to record this extra information and track the code module's progress as it completed an audit. Multiple logger instances were used to separate essential log messages, such as errors or warnings, and less important messages specific to a single audit table (see Figure 3 and the Implementation section). This separation organized the log messages into different files, making it easier to review later if needed.

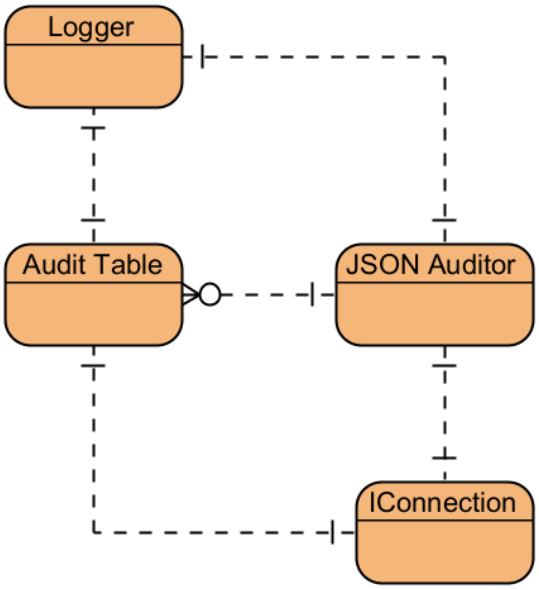


Figure 3. Relationships between classes.

One feature the code module needed was a way to ignore fields containing JSON data. While the records were in JSON format, some of the record's fields might have additional JSON (see Appendix A). These fields must be ignored and omitted from the audit (see item III in the Additional Requirements Summary section). Initially, this was done by attempting to detect which fields held JSON data. An issue arose where a field might only contain null values (called

"None" in Python). Null is valid for any data type (numbers, strings, etc.), so it cannot be determined if that field might actually hold JSON data. While it was unlikely that all the records would have a null value for the same field, the code module still needed to account for the possibility. This method was removed in favor of specifying a list of fields to ignore in the code module's configuration. While identifying a field to ignore is manual, it allows additional fields to be ignored if desired.

As mentioned above, the code module needed to have some configuration allowing options to be specified, such as fields to ignore, database connection parameters, location to save the log files, etc. A few options exist for storing the configuration, including in a JSON file or as variables in a Python file. The code module used a dictionary storing key-value pairs for this project's configuration. This had the advantage that the configuration could initially be stored as JSON due to how easily JSON can be deserialized in Python.

Initially, the code module stored the value of records' fields in the database as a string in JSON format. Therefore, a record could have a field with a null value, but a record could also be missing that field entirely. In both cases, the value would appear as null in the audit, but a distinction can be made in JSON. A JSON null can be used when the field exists but is null. This would appear in the audit database as a string containing "null." If the field does not exist, this can be shown by using a literal null in the audit database. Storing the value as JSON also perfectly preserves the original values since the original records were provided in JSON format. However, discussions with the project supervisor clarified that the values needed to be stored using applicable data types (see Table 1); this formed an additional requirement (see item IV in the Additional Requirements Summary section). Integers should be stored as integers, dates should be stored as dates, etc. This would improve performance when querying the database. In particular, representing dates as actual dates would allow date comparisons while avoiding string parsing when querying the database. This requirement specifically focused on the structure of the audit table.

Table 1. Python to MySQL data type mapping.

Python Data Type	**MySQL Data Type**
str (in datetime format)	DATETIME
str	LONGTEXT
int	BIGINT
float	DECIMAL(30, 10)
bool	BOOLEAN
None	LONGTEXT

The other requirements for the audit table structure include indexing a specified column (see item V in the Additional Requirements Summary section) and having a unique serial number for each audit record (see item VI in the Additional Requirements Summary section). Indexing a column, such as the primary key of the original record, would make lookups using that column faster. In cases where the indexed column is the primary key, all the audit records in the audit table with the same value for that column hold different versions of the same original record. This means querying for a record's history can be much faster. The other requirement, having a unique serial for each audit record, was accomplished by adding a primary key to the audit table. Note that the *audit* primary key (or serial number) is separate from the primary key of the original record. The serial number provides an easy way to identify each audit record, which can be used to distinguish between audit records for the same original record.

Additional Requirements Summary

Additional requirements were added to the project as the code module was being developed. Below is a summary of the added requirements mentioned in the Decisions section.

I. The table that stores the audit records (or "audit table") must be updated when the records' structure changes. If the records being audited were to have a field added, the audit must also have a field added. This requires changing the structure of the audit table to support the extra field. Appendix B explains the relationship between the original records (represented as JSON) and the audit records in the audit table.

II. Expanding on the requirement to log errors (see item IV in the Requirements section), the code module should also log any warnings, SQL commands, and other helpful information.

III. Fields containing JSON data should not be audited. This JSON represents related records. These related records could be considered records on a different table that are unnecessary to audit. If an audit of these related records is desired, it must be done separately.

IV. The fields holding data from the original records need to use equivalent SQL data types.

V. Provide indexing for a specified column. The original records being audited likely have a primary key; retrieving audit records will often use the original record's primary key.

VI. Uniquely identify each audit record using a serial number. While a feature in MySQL already does this (called "row number"), it can be more cumbersome.

Implementation

The final product of this project was a code module that fulfills the requirements described in the previous sections. This section will walk through how the code module audits the records given to it. The input records (original records) are passed to the code module in JSON format, so the first step is converting from JSON to the Python equivalent (see Appendix A). Some fields are ignored and excluded for the rest of the snapshot creation. Ignored fields are specified in the configuration.

The code module then runs several checks to ensure the integrity of the incoming records. These checks can help catch mistakes or invalid inputs.

1. Check if the list of records is empty. It is improbable that all the original records were deleted; this could indicate an issue with the program calling the code module.
2. Depending on the configuration, check to ensure all the original records have the same fields. While the input JSON is supposed to represent a table of records, JSON cannot enforce this.

Subsequent checks ensure the target audit table is prepared to receive the new audit records.

1. Check that the table specified exists in the audit database. If it does not exist and the configuration allows modifying the audit table, create a new table; otherwise, log an error and exit.
2. Check that the fields of the original records have not changed. If any fields have been added or removed (and the configuration allows modifying the audit table), rename (archive) the audit table and create a new audit table that includes the required fields.
3. Check that data types for each field on the audit table support the data types of the new audit records. If the audit table does not (and the configuration allows modifying the audit table), rename the audit table and create a new table that supports the incoming records (see Appendix C).

What occurs next depends on whether the configuration is set for a "full" or "partial" snapshot.

Full Snapshot

A full snapshot is the simplest method to audit records. It simply copies all the records into the audit, regardless of whether they have changed. The downside is that storing the audit takes much more space since many records will not change between snapshots. A full snapshot

is made by creating an audit record from the original record (see Appendix B) and inserting it into the audit table.

Partial Snapshot

A partial snapshot is much more space-efficient than a full snapshot since it only adds audit records for records that have changed since the last snapshot. This process starts by creating two temporary tables; all the new audit records are inserted into one of the tables, and the other table is populated with the most recent audit record for each original record. Next, the two tables are compared to determine which original records have been changed, removed, or added; these are then added to the audit table. Note that this ignores records that have not changed since the last snapshot, therefore saving space.

Future Work

While the project did produce a working code module, there are still some areas for improvement. This section will describe a few potential areas that could be improved further.

Currently, original records that have been deleted are represented by adding a new audit record with null used for the values of each field in place of the values that would have come from the original record. An example of how a deleted record would appear in the audit table is shown in Table 2. This method of representing a deleted record is ambiguous, as there could be an original record with only null values for each field. When its corresponding audit record is inserted into the audit table, it would appear to indicate that the record was deleted. One option to fix this would be to add an audit metadata field that clarifies what the audit record represents (i.e., a new record, deleted record, or modified record).

Table 2. Indication of deleted record in audit table
The blue text denotes the audit metadata fields while the green text represents the fields from the original record; see Appendix : From JSON to Audit.

audit_serial	audit_datetime	id	firstName	lastName	age
1	2024-01-01 16:35:00	123	Timothy	Zink	21
2	2024-02-01 16:35:00	NULL	NULL	NULL	NULL

Using a framework or library to handle interaction with the audit database was decided against in favor of implementing interactions from scratch. Utilizing such a framework could still be viable despite lacking more complex control over the database. This would require additional research into different libraries and redesigning the project to use the provided features. A couple of advantages of using a library for database interactions are improved readability and reliability.

This project implemented custom logging, which provides some basic logging features specifically designed for the code module. However, Python has a built-in logging module (Python Software Foundation, 2001-2024) for logging information and errors. It was not used because its existence was only realized after much of the code module was already implemented using the custom logger. A potential improvement would be replacing the custom logger with Python's built-in logging module.

References

(n.d.). (Stack Exchange) Retrieved from Stack Overflow: https://stackoverflow.com/questions/tagged/python

A Kenneth Reitz Project. (2024). *Developer Interface*. Retrieved from Requests: HTTP for Humans: https://requests.readthedocs.io/en/latest/

Becker, A. (n.d.). *HeidiSQL - MariaDB, MySQL, MSSQL, PostgreSQL and SQLite made easy*. Retrieved from HeidiSQL: https://www.heidisql.com/

Financial Modeling Prep. (2017-2024). *Free Stock Market API and Financial Statements API*. Retrieved from Financial Modeling Prep: https://site.financialmodelingprep.com/developer/docs

Introducing JSON. (n.d.). Retrieved from JSON: https://www.json.org/json-en.html

MariaDB. (2024). *Python to MariaDB Connector*. Retrieved from MariaDB: https://mariadb.com/resources/blog/how-to-connect-python-programs-to-mariadb/

Oracle. (2024). *MySQL Community Downloads*. (Oracle) Retrieved from MySQL: https://www.mysql.com/

Oracle. (2024). *MySQL Connector/Python Developer Guide*. Retrieved from MySQL: https://dev.mysql.com/doc/connector-python/en/

Python Software Foundation. (2001-2024). *3.12.2 documentation*. Retrieved from Python 3.12.2 documentation: https://docs.python.org/3/

Python Software Foundation. (2001-2024). *Logging facility for Python*. Retrieved from 3.12.2 documentation: https://docs.python.org/3/library/logging.html

Python Software Foundation. (2001-2024). *Welcome to Python*. Retrieved from Python Software Foundation: https://www.python.org/

Refsnes Data. (1999-2024). *JSON - Introduction*. Retrieved from W3Schools: https://www.w3schools.com/js/js_json_intro.asp

Refsnes Data. (1999-2024). *MySQL Tutorial*. Retrieved from W3Schools: https://www.w3schools.com/MySQL/default.asp

Refsnes Data. (1999-2024). *Python Tutorial*. Retrieved from W3Schools: https://www.w3schools.com/python/

SQLAlchemy. (2007-2024). *SQLAlchemy 2.0 Documentation*. Retrieved from SQLAlchemy: https://docs.sqlalchemy.org/en/20/

Tilley, S. (2020). *System Analysis and Design* (12th ed.). Boston: Cengage Learning.

van Rossum, G., Lehtosalo, J., & Langa, Ł. (2014, September 29). *PEP 484 – Type Hints*. Retrieved from Python Enhancement Proposals: https://peps.python.org/pep-0484/

Visual Paradigm. (2024). *Ideal Modeling & Diagramming Tool for Agile Team Collaboration*. Retrieved from Visual Paradigm: https://www.visual-paradigm.com/

Appendix

Appendix A: JSON and Python Equivalent

JavaScript Object Notation (JSON) is a format often used to serialize data. This appendix briefly describes JSON format and how it is deserialized in Python. There are six different data types in JSON; each is listed below with an example.

- Strings

```
"This is a string"
```

- Numbers

```
123
12.3
```

- Boolean

```
true
false
```

- Null

```
null
```

Both arrays and objects can hold more JSON data (even additional arrays or objects).

- Arrays

```
[
   "list",
   "of",
   "strings"
]
```

```
[
   123,
   456,
   789
]
```

```
[]
```

```
[
   {
      "myNumber": 123
   },
   {
      "myNumber": 456
   }
]
```

- Objects

```
{
   "myString": "string",
   "myNumber": 123,
   "myList": [ 1, 2, 3 ]
}
```

JSON data is stored as text (a string), which can be stored in a file (*.json) or deserialized. Python has a JSON module for handling JSON data. When deserializing a JSON string, the JSON data types are mapped to Python types:

JSON Data Type	Python Data Type	Notes
String	str	
Number	float or int	Depends on whether the number contains a decimal point
Boolean	bool	
Null	None	
Array	list	
Object	dict	dict is a Python dictionary

For example, some JSON data and the equivalent Python code for the same data are below. Note that it is very similar.

JSON:

```
[
   {
      "firstName": "Timothy",
      "lastName": "Zink",
      "age": 21,
```

```
      "height_meters": 1.9,
      "isFake": false,
   },
   {
      "firstName": "John",
      "lastName": "Doe",
      "age": null,
      "heightMeters": 1.8,
      "isFake": true
   }
]
```

Python:

```
[
   {
      "firstName": "Timothy",
      "lastName": "Zink",
      "age": 21,
      "height_meters": 1.9,
      "isFake": False,
   },
   {
      "firstName": "John",
      "lastName": "Doe",
      "age": None,
      "heightMeters": 1.8,
      "isFake": True
   }
]
```

Note that it is also possible for the original records to have fields that contain additional JSON data. This gets converted to the equivalent Python data structure but is later converted back into a JSON string before entering the audit table unless the field is ignored.

Appendix B: From JSON to Audit

This appendix describes the relationship of different fields as the original records go from their JSON representation to an audit record in an audit table. The JSON string provided to the code module represents a "table of records," the structure in JSON is then an array of objects. Each JSON object represents a single original record; shown is an example of a JSON object and the record it represents. Each field is color-coded to show which correspond to each other.

```
{
   "id": 123,
   "firstName": "Timothy",
   "lastName": "Zink",
   "age": 21,
   "heightMeters": 1.9,
   "isFake": False,
}
```

id	firstName	lastName	age	heightMeters	isFake
123	Timothy	Zink	21	1.9	F

After the record is deserialized in Python (see Appendix A), additional fields, called "metadata" or "audit metadata," are added, creating an "audit record" shown below. The "audit serial" is described in item VI in the Additional Requirements Summary section. The "audit datetime" is when the snapshot was taken (YYYY-MM-DD hh:mm:ss).

audit_serial	audit_datetime	id	firstName	lastName	age	heightMeters	isFake
1	2024-01-01 16:35:00	123	Timothy	Zink	21	1.9	F

The table above also reflects what would be in the audit table after a snapshot (assuming this was the first snapshot and there was only one incoming original record). If the original record were updated (for example, say my birthday occurred between two snapshots), the audit table would then look like this:

audit_serial	audit_datetime	id	firstName	lastName	age	heightMeters	isFake
1	2024-01-01 16:35:00	123	Timothy	Zink	21	1.9	F
2	2024-02-01 16:35:00	123	Timothy	Zink	22	1.9	F

This project initially stored the values of each field as a JSON string in the audit table. The table below shows how the fields would have appeared in MySQL if this had been used.

Note that each of the fields (except the metadata fields) is stored as text (strings); the quotes (") shown are literal quotes in the string. Also note that the strings stored in each of those fields are valid JSON strings (hence, "F" is "false").

audit_serial	audit_datetime	id	firstName	lastName	age	heightMeters	isFake
1	2024-01-01 16:35:00	123	"Timothy"	"Zink"	21	1.9	false

This method of storing data in the table is no longer used in favor of using the equivalent MySQL data types.

Appendix C: Python to SQL Data Type Mapping

Part of fulfilling the project requirement for robustness is automatically handling when the fields of the original records change (fields added/removed or the data type changes). Handling these changes is described in the Implementation section; this section will explain how converting between Python and SQL data types is dealt with.

The original records (coming from being in JSON format) have these six possible data types in Python: str (string), float, int (integer), bool (boolean), None, list, and dict (dictionary). If a field's value is a list or dictionary, the field initially held JSON data; these fields are converted strings (unless ignored). The table below shows how the data types in Python are mapped to MySQL data types.

Python Data Type	MySQL Data Type
str (in datetime format)	DATETIME
str	LONGTEXT
int	BIGINT
float	DECIMAL(30, 10)
bool	BOOLEAN
None	LONGTEXT

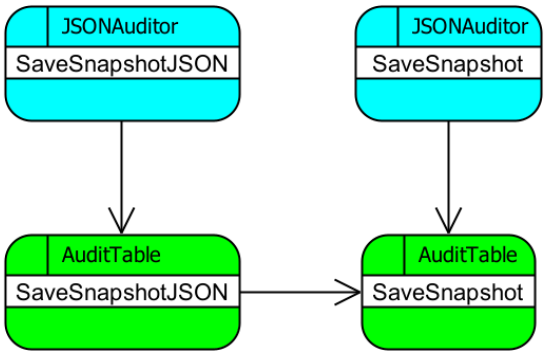
Note that None (Python's null value) is mapped to the same SQL data type as a string. This is because None could be in place of any other data type; string is used because it can support any other data type. That said, the data type for each field is determined in the context of all the incoming original records. If one record has a null value in a particular field while another record has a non-null value in the same field, the non-null value is used to determine the data type of that field.

A field is determined to be a datetime if it holds a string that can be parsed as a datetime based on the format in the configuration. In some cases, the data type might be "expanded." For example, a field on one record is an integer, while on another record, it is a float; the data type used would be float since it can also handle integers.

As mentioned, string is used when the data type of a field cannot be determined; string is also used if there are unresolved conflicts between data types. For example, if a record's field is a boolean while another record's same field is a float; the resulting data type used would fall back to a string. Note that while it is possible to store booleans as floats or integers (using 0 or 1), it was decided to avoid this to preserve the original value better.

Appendix D: "SaveSnapshot" Disambiguation

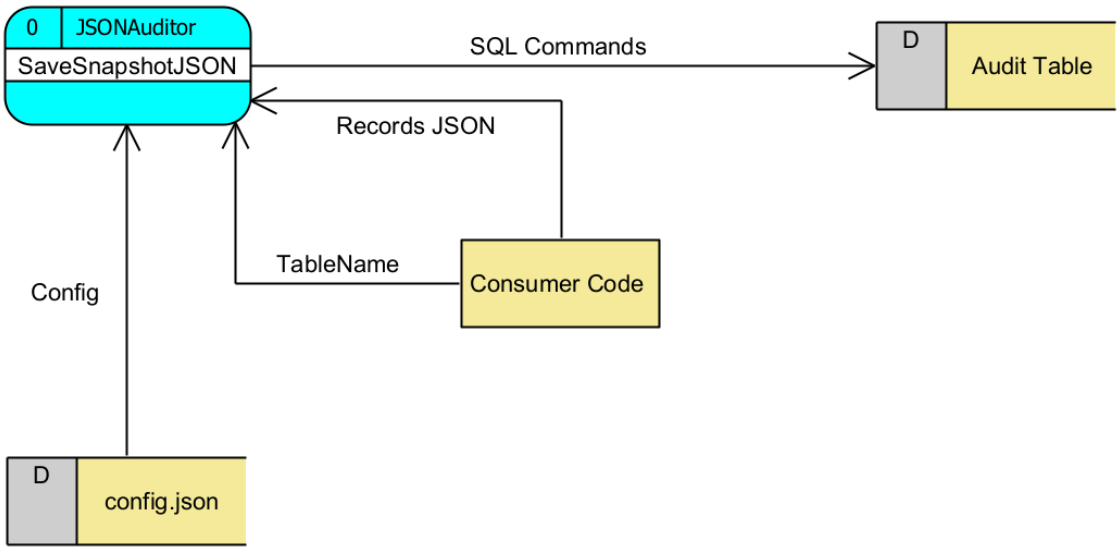
In the code module, there are two classes, "JSONAuditor" and "AuditTable," both of which have the methods "SaveSnapshot" and "SaveSnapshotJSON." A JSONAuditor object acts as a wrapper around one or more AuditTable objects. Calling methods "SaveSnapshot" or "SaveSnapshotJSON" on JSONAuditor calls the corresponding method on the specified AuditTable. Shown to the right is what method is called internally by other methods.



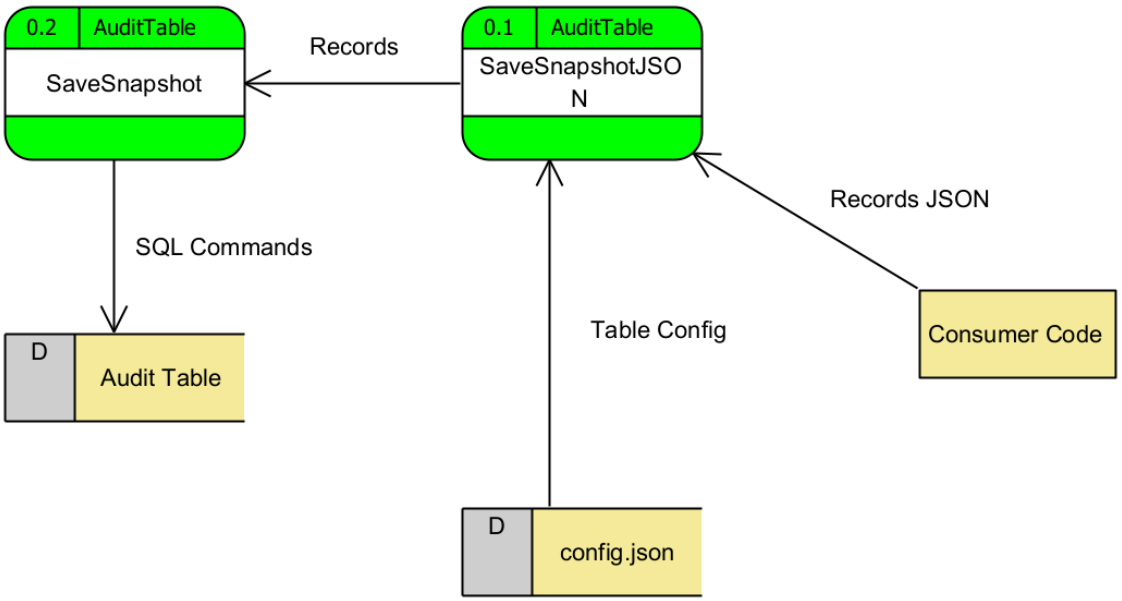
To reiterate, both of the methods on JSONAuditor act to select one of the AuditTable objects; additionally, "SaveSnapshotJSON" provides deserializing from JSON to the Python equivalent.

Appendix E: Data Flow Diagrams

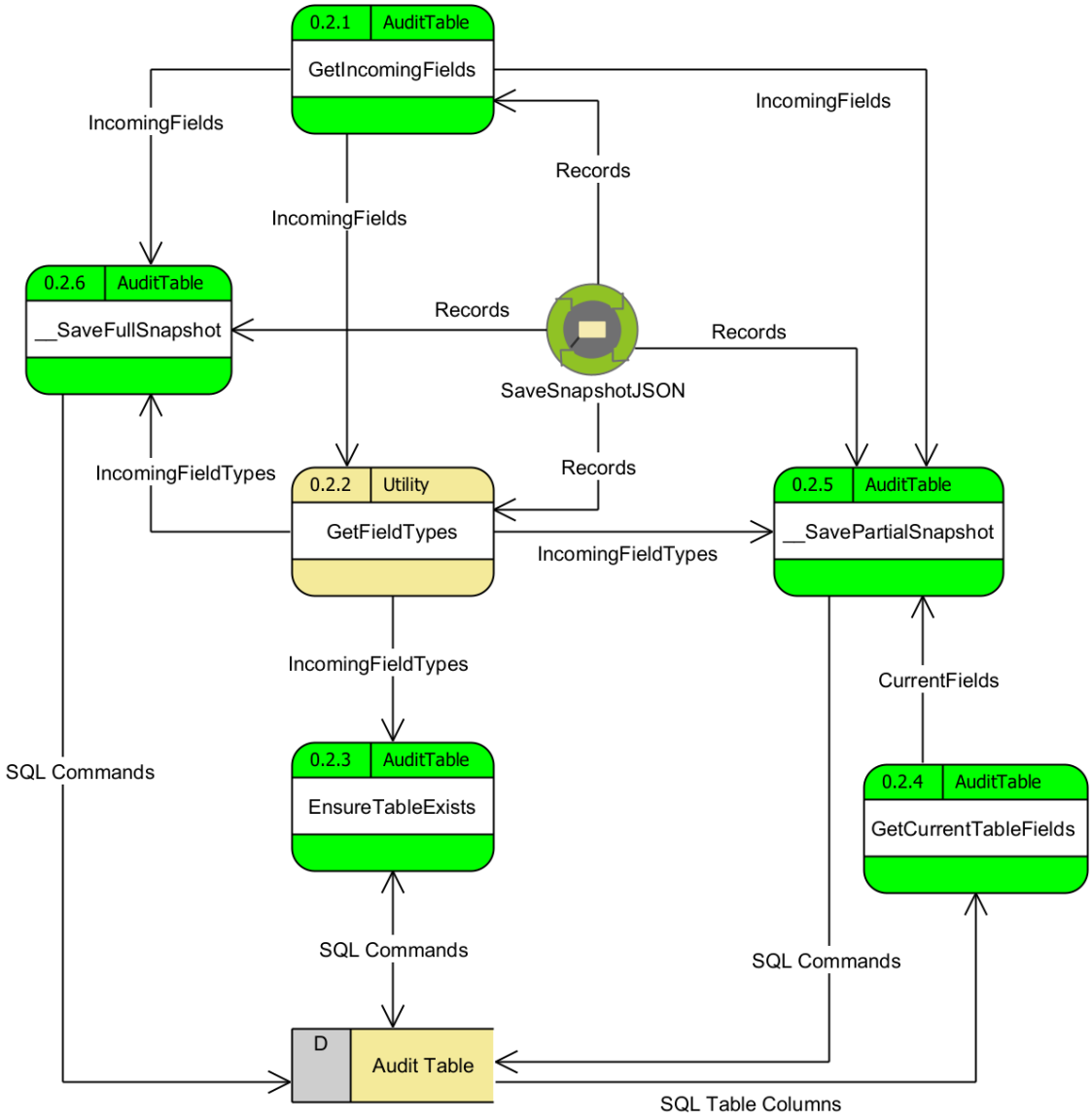
Context Diagram



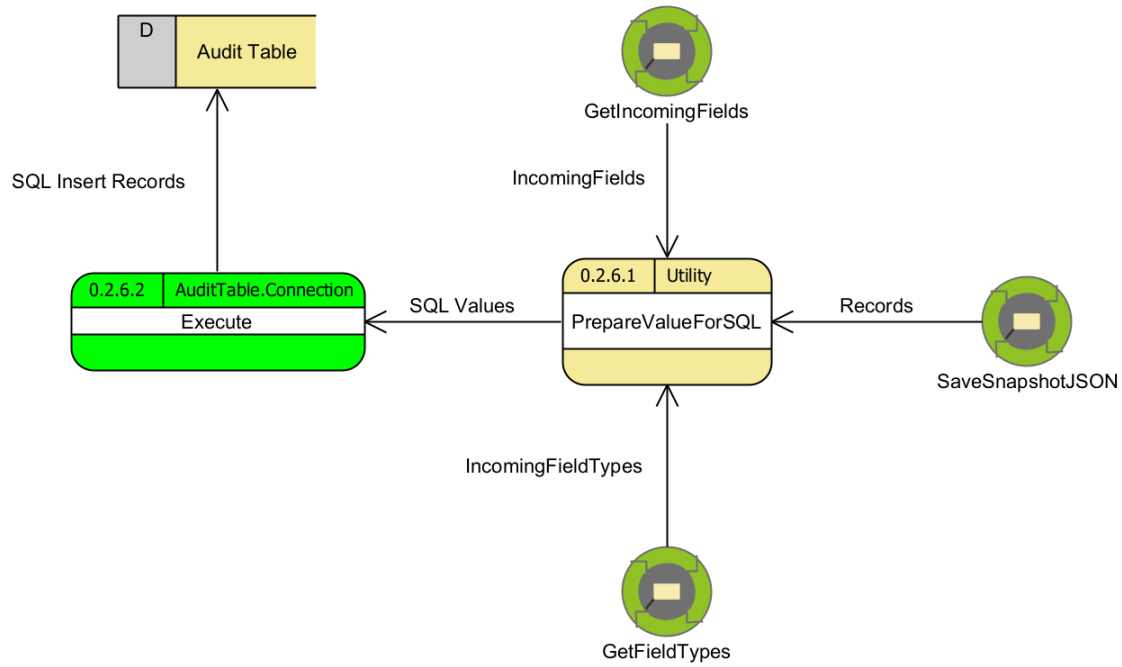
JSONAuditor.SaveSnapshotJSON



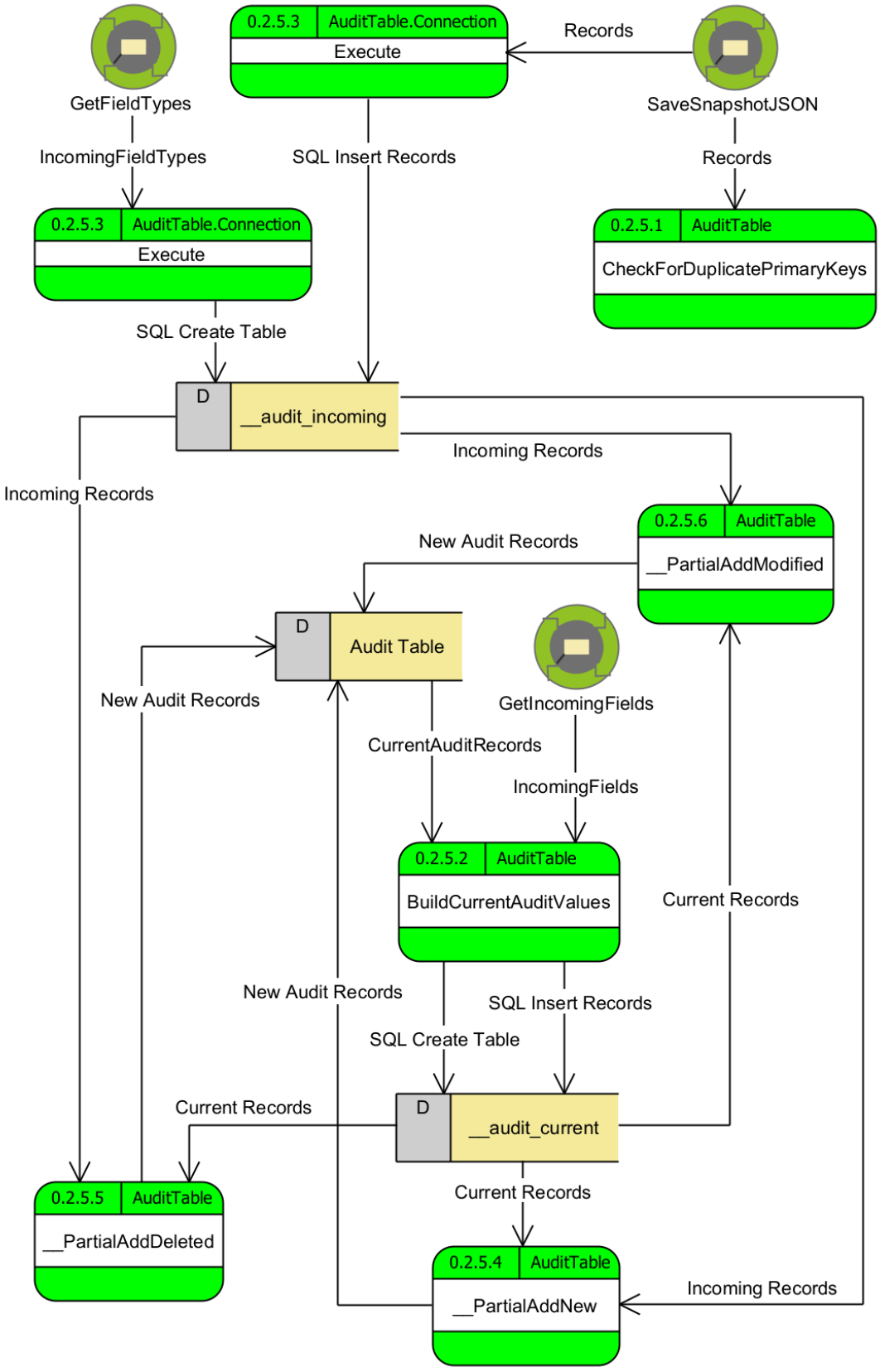
AuditTable.SaveSnapshot



AuditTable.__SaveFullSnapshot

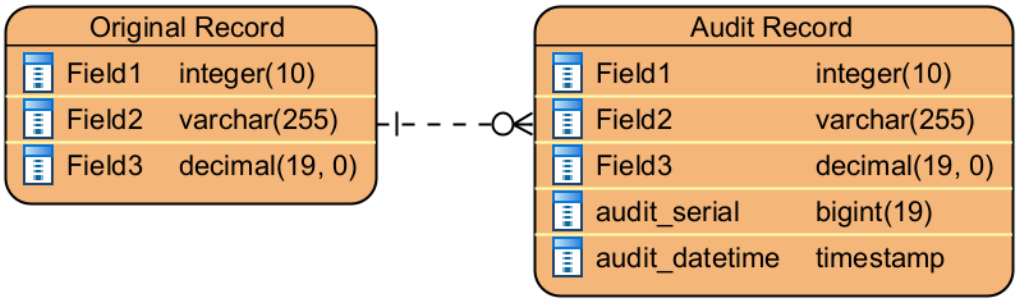


AuditTable.__SavePartialSnapshot

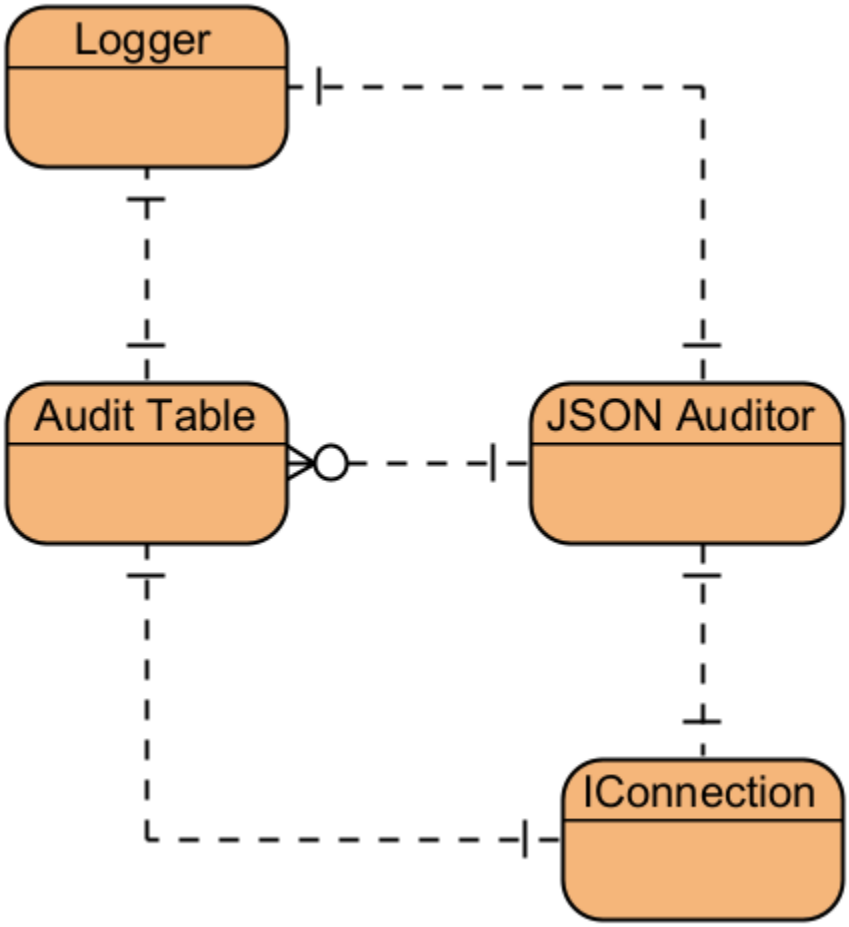


Appendix F: Entity Relationship Diagrams

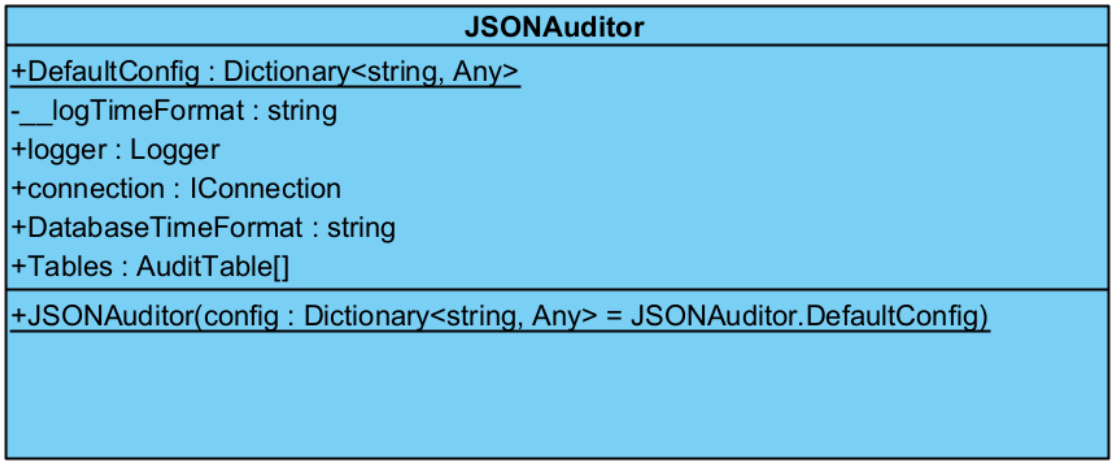
Original and Audit Records



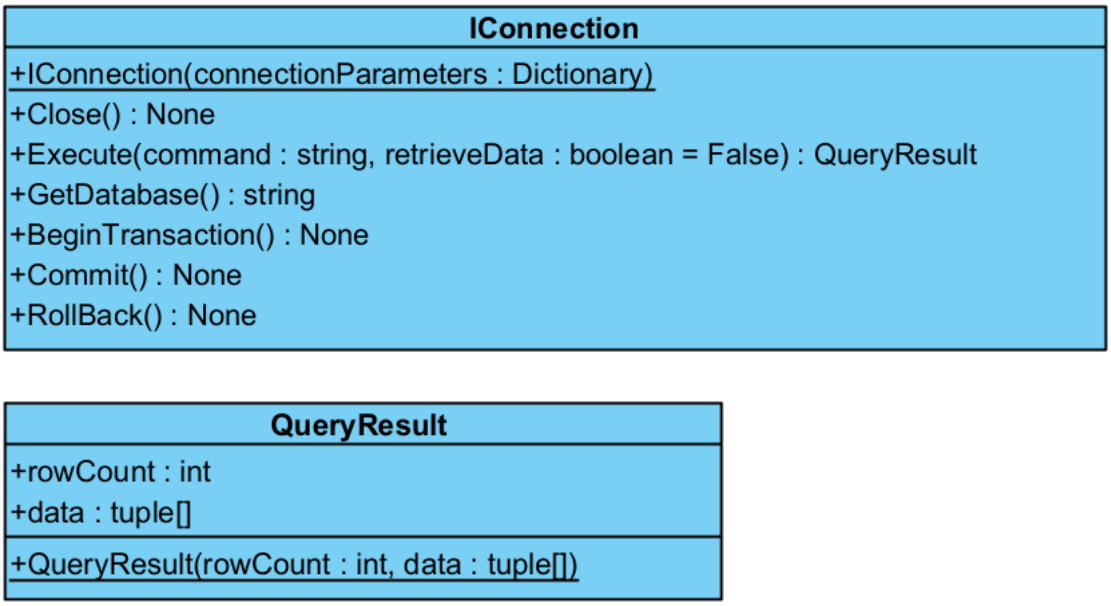
JSON Auditor Classes



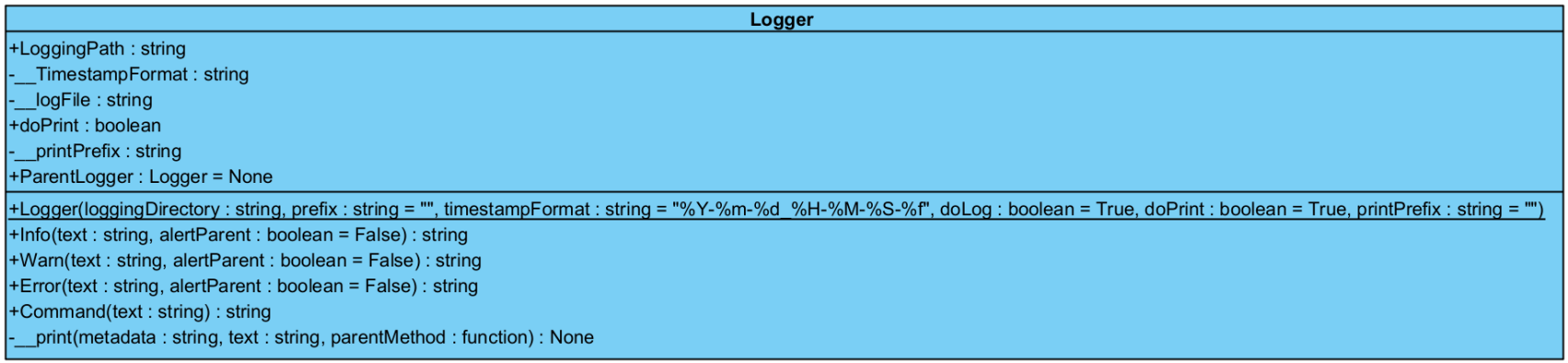
Appendix G: Class Diagrams




AuditTable
+DefaultConfig : Dictionary<string, Any>
+Logger : Logger
+Connection : IConnection
-__databaseTimeFormat : string
+AuditTableName : string
+AuditColumns : Dictionary<string, string
+StrictRowFields : boolean
+CreateTableIfNeeded : boolean
+AllowModifyTable : boolean
+DoPartialSnapshots : boolean
+PrimaryKeyField : string
+IndexPrimaryKeyField : boolean
+RecordTimeFormat : string
+IgnoreFields : string[]
+UseTransactions : boolean
+InsertsPreTransaction : int
+BatchInsertSize : int
+DeleteSnapshotOnFail : boolean
+CurrentSnapshotTimestamp : string = None
-__audit_current_name : string
-__audit_incoming_name : string
+AuditTable(connection : IConnection, databaseTimeFormat : string, config : Dicti...
+CreateTable(fieldsWithTypes : Dictionary<string, type>, audit_serial_start : int) : ...
+TableExists(tableName : string) : boolean
+ArchiveTable() : None
+EnsureTableExists(requiredFieldsWithTypes : Dictionary<string, type>) : None
+GetCurrentTableFields() : Dictionary<string, string>
+GetIncomingFields(incoming_rows : Dictionary<string, Any>[]) : string[]
+SaveSnapshot(incoming_rows : Dictionary<string, Any>[]) : tuple<boolean, int, in...
+SaveSnapshotJSON(jsonString : string) : tuple<boolean, int, int, int, int>
+DeleteSnapshot(timestamp : string) : None
+HasIndex(indexName : string) : boolean
+CheckForDuplicatePrimaryKeys(incoming_rows : Dictionary<string, Any>[], Prim...
+GetAuditHistory(primaryKeyFieldName : string, primaryKeyValue : Any, tableFiel...
+BuildCurrentAuditValues(tableFieldNames : string[], primaryKeyFieldName : strin...
-__SaveFullSnapshot(incoming_rows : Dictionary<string, Any>[], incomingFields : ...
-__SavePartialSnapshot(incoming_rows : Dictionary<string, Any>[], incomingField...
-__PartialAddNew(currentTableFields : string[], primaryKeyFields : string, timesta...
-__PartialAddDeleted(currentTableFields : string[], primaryKeyField : string, timest...
-__PartialAddModified(currentTableFields : string[], primaryKeyField : string, times...


IConnection
+IConnection(connectionParameters : Dictionary)
+Close() : None
+Execute(command : string, retrieveData : boolean = False) : QueryResult
+GetDatabase() : string
+BeginTransaction() : None
+Commit() : None
+RollBack() : None

QueryResult
+rowCount : int
+data : tuple[]
+QueryResult(rowCount : int, data : tuple[])


Logger
+LoggingPath : string
-__TimestampFormat : string
-__logFile : string
+doPrint : boolean
-__printPrefix : string
+ParentLogger : Logger = None
+Logger(loggingDirectory : string, prefix : string = "", timestampFormat : string = "%Y-%m-%d_%H-%M-%S-%f", doLog : boolean = True, doPrint : boolean = True, printPrefix : string = "")
+Info(text : string, alertParent : boolean = False) : string
+Warn(text : string, alertParent : boolean = False) : string
+Error(text : string, alertParent : boolean = False) : string
+Command(text : string) : string
-__print(metadata : string, text : string, parentMethod : function) : None