GRAM Specification (v1.0)

The Formal Standard for Graph-Rule Analytical Mapping—Deterministic Machine-Reasoning over Relational Structures

Ormle

2026-03-01

Table of contents

Status: Published · **Category:** Technical Specification · **Publisher:** Ormle · **GitHub:** ormle-dev/ormle-standard · **Site:** ormle.com/Standard

---

Abstract

Large language models are increasingly employed for text-to-SQL generation, yet they operate on SQL Data Definition Language (DDL)—physical metadata that describes *how* data is stored, not *what* it means. This forces the model to infer join semantics, cardinality, participation constraints, and column roles from naming conventions alone, a process that is the primary source of errors in generated SQL.

This document defines the **GRAM Specification (v1.0)**—a formal standard for representing relational database schemas as collections of **Asserted Predicates** rather than physical storage descriptors. Grounded in Object-Role Modeling (Halpin, 2006) and formalized with predicate calculus, **GRAM** (Graph-Rule Analytical Mapping) provides a semantic layer that captures an organization's unique business grammar—the rules, relationships, and vocabulary that define how its data means what it means—bridging the gap between physical DDL and the conceptual schema required for deterministic machine reasoning.

By formalizing entities, fact types, reference schemes, functional dependencies, existence constraints, data types, enumerated domains, alternate readings, sample values, and role names as first-class citizens of the schema, the GRAM specification enables large language models and automated systems to determine correct join semantics, column operations, and domain boundaries without inferential guesswork. Eliminating structural inference from the reasoning pipeline reduces the error surface to semantic misinterpretation alone, removing the class of errors caused by ambiguous join paths, missing cardinality information, and unnamed foreign key relationships.

---

1. Introduction: The Semantic Gap

Modern large language models (LLMs) exhibit systematic failure in text-to-SQL tasks because SQL Data Definition Language (DDL) constitutes **physical metadata**: it describes *how* data is stored, not *what* it means. A column definition such as `CustomerID INT NOT NULL` is a physical constraint. It provides no semantic information about the role that `CustomerID` plays in the business domain.

GRAM addresses this deficiency by representing a database as a collection of **Fact Types**—binary relations expressed as natural-language readings—augmented with formally defined constraints. This representation allows an LLM to *read* the schema rather than *infer* it, moving from stochastic interpretation to deterministic reasoning.

---

2. Definitions and Notation

2.1 Entities and Value Types

Let $\mathcal{E}$ denote the set of all **Entity Types** and $\mathcal{V}$ denote the set of all **Value Types**. An Entity Type represents a non-value object in the domain (e.g., `Employee`, `Order`). A Value Type represents a terminal property (e.g., `FirstName`, `OrderDate`).

An Entity Type may optionally carry a **namespace** that represents the database schema qualifier (e.g., `dbo`, `Sales`). When present, the fully-qualified table name is `namespace.name`. Two entities sharing the same name but differing in namespace are considered distinct types.

2.2 Fact Types

A **Fact Type** is a binary relation R defined by a predicate P over two participants:

$$R = \{(e_1, e_2) \mid P(e_1, e_2) \text{ is True}\}$$

where $e_i \in (\mathcal{E} \cup \mathcal{V})$. Version 1.0 of the GRAM specification normatively defines **binary fact types (arity 2)**, which represent relationships between two participants expressed as natural-language readings (e.g., *"Employee handles Order"*). This includes both entity-to-entity relationships and entity-to-value-type properties (e.g., *"Employee has FirstName"*). These readings constitute the **business grammar** of the domain—the precise vocabulary and sentence structures through which the organization's data tells its own story.

Each specific instance in the database—a **tuple**—is a formal assertion that the predicate defined by the **Reading** is true for a given pair of participants.

2.3 Roles

Each participant in a Fact Type occupies a **Role**. A role binds an entity or value type to a specific position within the predicate. Roles carry constraint metadata (uniqueness, mandatoriness) that governs the logical behavior of the relation.

2.4 Reference Schemes

A **Reference Scheme** defines how instances of an entity type are uniquely identified. It is the preferred identifier—analogous to a primary key in a relational database. Formally, for each entity type $E \in \mathcal{E}$, the reference scheme defines an injective function:

$$\text{ref} : E \to D_{\text{id}} \quad \text{such that} \quad \forall a, b \in E : \text{ref}\,(a) = \text{ref}\,(b) \implies a = b$$

where D_{id} is the identifier domain (e.g., integer employee numbers, string codes). In a GRAM-compliant schema, the reference scheme is declared using the notation `Employee(empId)`, rendering as *(empId)* beneath the entity name on the diagram.

Each entity has exactly one reference scheme. Additional attributes requiring uniqueness (e.g., email address) are modeled as separate fact types with uniqueness constraints, making them **alternate keys**.

The reference scheme is critical for machine reasoning: it tells an LLM which column to use in `JOIN ON` clauses and `WHERE` filters. Without it, the model must guess from column names whether `EmpID`, `EmployeeNumber`, or `EmpCode` is the correct join key.

2.5 Data Types

Value types in the GRAM specification carry an explicit `valueDataType` property that declares the semantic data category. The specification defines five normative data types:

Data Type	SQL Affinity	LLM Significance
Text	VARCHAR / NVARCHAR	String comparison, LIKE patterns, concatenation
Number	INT / DECIMAL / FLOAT	SUM, AVG, arithmetic, ORDER BY numeric
Date	DATE / DATETIME	Date functions, BETWEEN, DATEADD/DATEDIFF
Boolean	BIT	Predicate filtering (WHERE IsActive = 1)

Data Type	SQL Affinity	LLM Significance
Currency	MONEY / DECIMAL	SUM, AVG with monetary formatting, rounding rules

DDL alone forces an LLM to infer column semantics from names like `amt` or `dt`. The explicit data type eliminates this inference, enabling the model to select appropriate SQL functions and aggregation strategies.

2.6 Enumerated Types

An **Enumerated Type** declares a closed set of allowed values for an object type. The GRAM specification supports enumeration on both value types and entity types, serving distinct purposes.

Value type enums constrain terminal properties. The enumerated values define the complete domain of a column. For example, an `OrderStatus` value type with values `{Pending, Shipped, Delivered}` tells the LLM the exact set of valid filter values for status queries.

Entity type enums are **closed-world assertions** on entities that participate in relationships. For example, a `Department` entity with values `{Engineering, Marketing, Sales, HR}` is not merely a constrained string—it is a first-class entity referenced by other facts (e.g., *"Employee works for Department"*, *"Department has Budget"*). The enumeration tells the LLM that there are exactly four departments, enabling precise `WHERE ... IN (...)` clauses and exhaustive groupings.

In both cases, the complete set of allowed values is exported in the schema as `enumValues`, providing the LLM with a closed-world guarantee rather than an open-ended guess.

2.7 Alternate Readings

An **Alternate Reading** is an additional natural-language phrasing of the same fact type. While the primary reading and inverse reading define the canonical predicate, alternate readings provide **synonymous expressions** that increase the surface area for question-to-relation matching.

For example, a fact type with primary reading *"Order has OrderDetail"* might carry an alternate reading *"Order contains OrderDetail."* When an LLM receives a user question referencing "line items contained in an order," the alternate reading provides a direct lexical match that the primary reading alone would not.

Alternate readings are purely semantic metadata—they do not alter the relational structure, constraints, or role bindings. They are exported as an array of strings alongside the primary and inverse readings.

2.8 Sample Values

Sample Values are concrete example instances attached to an entity type (e.g., `"Acme Corp"`, `"Globex Inc"` for a `Company` entity). They are exported in the schema as `sampleValues`.

Sample values serve two purposes for downstream reasoning:

- **Disambiguation:** They clarify what an entity represents in the real world. An entity named `Item` could be a product, a menu item, or an invoice line. Sample values like `"Widget A"`, `"Gizmo Pro"` immediately resolve the ambiguity.
- **Query grounding:** When an LLM generates example queries or validates output, concrete values provide realistic test data without requiring database access.

2.9 Role Names

A **Role Name** is an explicit override for the foreign key column name produced when a binary fact type maps to a relational schema. Ordinarily, when entity B is referenced from entity A's table, the FK column is named after B's `referenceScheme`. A role name replaces this default:

$$\text{FK column} = \begin{cases} \text{roleName} & \text{if declared} \\ B.\,\text{referenceScheme} & \text{otherwise} \end{cases}$$

Role names are necessary in three situations:

1. **Renamed foreign keys.** The child table uses a different column name than the parent's primary key. Example: `EventLog.ActorId` references `Identity.IdentityId`.
2. **Multiple references to the same entity.** A single table references the same parent more than once. Example: `Flight` has both `DepartureAirportCode` and `ArrivalAirportCode`, both referencing `Airport.AirportCode`.
3. **Self-referential relationships.** An entity references itself through a distinctly named column. Example: `Project.ParentProjectId` references `Project.ProjectId`.

---

3. The Predicate Calculus of GRAM

3.1 Atomic Predicates

When a user defines a fact in a GRAM graph, they define the **membership criteria** for a relation. Each fact's `reading` property constitutes an **atomic predicate**. For a binary fact type between entities E_1 and E_2:

$$P(x, y) \quad \text{where} \quad x \in E_1, y \in E_2$$

GRAM Property	Formal Definition	Significance
`reading`	Atomic Predicate $P(x, y)$	Defines the semantic truth of the relation
`entityName`	Domain $E_i \in (\mathcal{E} \cup \mathcal{V})$	Restricts each role to a declared entity or value type
`isUnique`	Functional Dependency $X \to Y$	Defines cardinality and uniqueness
`isMandatory`	Existence Constraint $\forall x\, \exists y$	Defines total participation

3.2 Inverse Readings

For each binary Fact Type, the GRAM specification explicitly captures the **Inverse Reading**, ensuring that the relation is not a directed pointer (as in a Foreign Key) but a **bi-directional logical assertion**. If the forward reading is *"Employee handles Order"*, the inverse is *"Order is handled by Employee"*. This duality enables an LLM to traverse the graph in either direction without ambiguity.

---

4. Constraint Formalization

Constraints in the GRAM specification are not auxiliary metadata; they are **first-class logical laws** governing the behavior of each relation.

4.1 Uniqueness Constraints and Functional Dependencies

The `isUnique` property on a role defines a constraint on the relation. If role r_1 is unique in a binary relation between E_1 and E_2, we have a function $f : E_1 \to E_2$:

$$\forall x \in E_1, \forall y, z \in E_2 : \big(\; P(x, y) \land P(x, z) \; \big) \implies y = z$$

This is not a database configuration option; it is a **logical law**. When an LLM interprets a GRAM-compliant graph, it does not guess join cardinality—it computes the **deterministic path** of the query. By capturing uniqueness at the role level, the specification explicitly identifies the **candidate keys** of every relation.

4.2 Mandatoriness and Existence Constraints

The `isMandatory` property defines an **existence dependency**. If entity E_1 must participate in predicate P:

$$\forall x \in E_1 \exists y \in E_2 : P(x, y)$$

An LLM receiving this constraint knows to generate an `INNER JOIN`. Without it, the appropriate operation is a `LEFT JOIN`. This distinction is the difference between a correct result set and a missing-row error.

4.3 Cardinality Classification

The combination of uniqueness constraints on roles in a binary fact type determines the cardinality of the underlying relation:

Role 1 Unique	Role 2 Unique	Cardinality	Interpretation
No	No	$N : M$	Many-to-many; relation is irreducible
Yes	No	$1 : N$	Functional dependency; one side determines the other
No	Yes	$N : 1$	Functional dependency (reverse direction)
Yes	Yes	$1 : 1$	Bijection; both sides determine each other

5. The Ontological Failure of SQL DDL

Conventional metadata management suffers from an **ontological collapse**. By treating DDL as a sufficient semantic source, practitioners commit a category error. DDL defines the **physical schema**—the mechanism of storage—but remains silent on the **conceptual schema**—the nature of the facts being stored.

This distinction is not new. Codd (1970) argued for separating data representation from storage; Chen (1976) formalized it with the Entity-Relationship model; Halpin (1989–2006) advanced it with Object-Role Modeling, grounding conceptual schemas in natural-language predicates. Yet the gap persists—a `FOREIGN KEY` constraint in SQL enforces referential integrity but does not express the predicate. Consider:

```
ALTER TABLE Orders
    ADD CONSTRAINT FK_Emp
    FOREIGN KEY (EmpID) REFERENCES Employees(ID);
```

This statement tells the database engine *"do not break the link."* It provides no information about what the link *means*—whether an employee *handles*, *approves*, or *delivers* the order. The GRAM specification resolves this by binding every relation to an explicit natural-language predicate.

---

6. Normative Comparison

The following comparison illustrates the difference between physical metadata and a GRAM-compliant semantic representation for the query: *"How many orders were handled by each employee?"*

6.1 DDL-Only Interpretation

An LLM operating on DDL alone observes a nullable `EmpID` column in the `Orders` table. It must infer join semantics, potentially selecting a `LEFT JOIN` when an `INNER JOIN` is correct, miscounting null entries, or double-counting in misidentified many-to-many relationships.

6.2 GRAM-Compliant Interpretation

An LLM receiving the GRAM graph reads:

> "Employee handles Order. Each Order must be handled by exactly one Employee."

The mandatoriness constraint ($\forall$ Order $\exists$ Employee) dictates an `INNER JOIN`. The uniqueness constraint on the Order role establishes the functional dependency Order $\rightarrow$ Employee, confirming a $1 : N$ cardinality. The generated query is deterministic:

```
SELECT e.EmployeeName, COUNT(o.OrderID) AS OrderCount
FROM Employees e
INNER JOIN Orders o ON o.EmpID = e.ID
GROUP BY e.EmployeeName;
```

---

7. Schema Representation

A GRAM-compliant schema is serialized as a JSON document. The following fragment illustrates a representative subset of the schema, exercising every normative feature: entities with reference schemes and sample values, an optional `namespace` for database schema qualification (applicable only to non-value-type entities), value types spanning all five data types, a value type enum, an entity enum, a self-referential relationship with a role name, multiple references to the same entity, an alternate reading, and varied constraint patterns.

```
{
  "entities": [
    {
      "name": "Employee",
      "namespace": "dbo",
      "referenceScheme": "empId",
      "isValueType": false,
      "isEnum": false,
      "sampleValues": ["Alice Chen", "Bob Martinez"]
    },
    {
      "name": "Order",
      "referenceScheme": "orderId",
      "isValueType": false,
      "isEnum": false,
      "sampleValues": ["ORD-1001", "ORD-1002"]
    },
    {
      "name": "Project",
      "referenceScheme": "projectId",
      "isValueType": false,
      "isEnum": false,
      "sampleValues": ["Apollo", "Horizon"]
    },
    {
      "name": "Department",
      "referenceScheme": "code",
      "isValueType": false,
      "isEnum": true,
      "enumValues": ["Engineering", "Marketing", "Sales", "HR"]
    },
    {
      "name": "FirstName",
      "isValueType": true,
      "valueDataType": "Text"
    },
    {
      "name": "OrderDate",
```

```
      "isValueType": true,
      "valueDataType": "Date"
    },
    {
      "name": "OrderTotal",
      "isValueType": true,
      "valueDataType": "Currency"
    },
    {
      "name": "HeadCount",
      "isValueType": true,
      "valueDataType": "Number"
    },
    {
      "name": "IsActive",
      "isValueType": true,
      "valueDataType": "Boolean"
    },
    {
      "name": "OrderStatus",
      "isValueType": true,
      "valueDataType": "Text",
      "isEnum": true,
      "enumValues": ["Pending", "Shipped", "Delivered", "Cancelled"]
    }
  ],
  "factTypes": [
    {
      "reading": "Employee handles Order",
      "inverseReading": "Order is handled by Employee",
      "alternateReadings": ["Employee processes Order"],
      "arity": 2,
      "roles": [
        {
          "entityName": "Employee",
          "isUnique": false,
          "isMandatory": false
        },
        {
          "entityName": "Order",
          "isUnique": true,
          "isMandatory": true
        }
      ]
    },
    {
```

```
      "reading": "Employee submits Order",
      "inverseReading": "Order is submitted by Employee",
      "arity": 2,
      "roles": [
        {
          "entityName": "Employee",
          "roleName": "SubmittedById",
          "isUnique": false,
          "isMandatory": false
        },
        {
          "entityName": "Order",
          "isUnique": true,
          "isMandatory": true
        }
      ]
    },
    {
      "reading": "Employee has FirstName",
      "inverseReading": "FirstName is of Employee",
      "arity": 2,
      "roles": [
        {
          "entityName": "Employee",
          "isUnique": false,
          "isMandatory": true
        },
        {
          "entityName": "FirstName",
          "isUnique": false,
          "isMandatory": true
        }
      ]
    },
    {
      "reading": "Employee has IsActive",
      "inverseReading": "IsActive is of Employee",
      "arity": 2,
      "roles": [
        {
          "entityName": "Employee",
          "isUnique": false,
          "isMandatory": true
        },
        {
          "entityName": "IsActive",
```

```
        "isUnique": true,
        "isMandatory": true
      }
    ]
  },
  {
    "reading": "Employee works for Department",
    "inverseReading": "Department employs Employee",
    "arity": 2,
    "roles": [
      {
        "entityName": "Employee",
        "isUnique": false,
        "isMandatory": true
      },
      {
        "entityName": "Department",
        "isUnique": false,
        "isMandatory": false
      }
    ]
  },
  {
    "reading": "Order has OrderDate",
    "inverseReading": "OrderDate is of Order",
    "arity": 2,
    "roles": [
      {
        "entityName": "Order",
        "isUnique": false,
        "isMandatory": true
      },
      {
        "entityName": "OrderDate",
        "isUnique": true,
        "isMandatory": true
      }
    ]
  },
  {
    "reading": "Order has OrderTotal",
    "inverseReading": "OrderTotal is of Order",
    "arity": 2,
    "roles": [
      {
        "entityName": "Order",
```

```
        "isUnique": false,
        "isMandatory": true
      },
      {
        "entityName": "OrderTotal",
        "isUnique": true,
        "isMandatory": true
      }
    ]
  },
  {
    "reading": "Order has OrderStatus",
    "inverseReading": "OrderStatus is of Order",
    "arity": 2,
    "roles": [
      {
        "entityName": "Order",
        "isUnique": false,
        "isMandatory": true
      },
      {
        "entityName": "OrderStatus",
        "isUnique": true,
        "isMandatory": true
      }
    ]
  },
  {
    "reading": "Department has HeadCount",
    "inverseReading": "HeadCount is of Department",
    "arity": 2,
    "roles": [
      {
        "entityName": "Department",
        "isUnique": false,
        "isMandatory": false
      },
      {
        "entityName": "HeadCount",
        "isUnique": true,
        "isMandatory": false
      }
    ]
  },
  {
    "reading": "Project is sub-project of Project",
```

```
      "inverseReading": "Project has sub-project Project",
      "arity": 2,
      "roles": [
        {
          "entityName": "Project",
          "isUnique": false,
          "isMandatory": false
        },
        {
          "entityName": "Project",
          "roleName": "ParentProjectId",
          "isUnique": false,
          "isMandatory": false
        }
      ]
    }
  ]
}
```

In this representation:

- **Reference schemes.** The `referenceScheme` on Employee (`"empId"`), Order (`"orderId"`), and Project (`"projectId"`) identifies the join key column for each entity.
- **Namespace.** The optional `namespace` on Employee (`"dbo"`) specifies the database schema that qualifies the table name (e.g., `dbo.Employee`). When present, the fully-qualified name is `namespace.name`. This field is only applicable to non-value-type entities; value types inherit context from their owning entity. Entities with different namespaces but the same name (e.g., `Sales.Customer` and `Finance.Customer`) are treated as distinct.
- **Functional dependency.** The uniqueness of the Order role in "Employee handles Order" establishes Order → Employee (each order is handled by exactly one employee).
- **Total participation.** The mandatoriness of the Order role asserts ∀ Order, ∃ Employee, dictating an `INNER JOIN`.
- **Alternate readings.** The `alternateReadings` array on "Employee handles Order" provides synonymous phrasings ("Employee processes Order") for question-to-relation matching.
- **All five data types.** Value types demonstrate every normative data type: `Text` (FirstName), `Date` (OrderDate), `Currency` (OrderTotal), `Number` (HeadCount), and `Boolean` (IsActive). Each type directs the LLM to the appropriate SQL functions and aggregation strategies.
- **Value type enum.** OrderStatus declares `enumValues` with a `valueDataType` of `Text`, giving the LLM a closed set of valid filter values for status queries.
- **Entity enum.** Department declares `enumValues` as a first-class entity referenced by other facts, enabling precise `WHERE ... IN (...)` clauses and exhaustive groupings.
- **Sample values.** Concrete instances on Employee, Order, and Project disambiguate what each entity represents and provide realistic test data.
- **Self-referential relationship.** "Project is sub-project of Project" references the same entity in both roles. The `roleName` of `"ParentProjectId"` disambiguates the FK column, which would otherwise collide with the entity's own `projectId`.

- **Cross-entity role name.** "Employee submits Order" creates a second FK from Order to Employee. The `roleName` of `"SubmittedById"` overrides the default column name (`empId`), letting the LLM distinguish the two relationships without ambiguity.
- **Varied constraint patterns.** The example includes mandatory-both-sides properties (FirstName, OrderDate, OrderTotal, OrderStatus), a mandatory-one-side entity relationship (Employee works for Department), and fully optional relationships (Department has HeadCount, Project self-ref), illustrating how constraint metadata governs JOIN type selection.

---

8. Conformance

The JSON structure defined in Section 7 constitutes the **normative serialization format** of the GRAM specification. Any system that produces or consumes schemas in this format is subject to the conformance requirements below.

A system is **GRAM-conformant** if it satisfies the following requirements:

1. Every relation in the schema is expressed as a Fact Type with an explicit natural-language reading.
2. Binary Fact Types include both the forward reading and the inverse reading.
3. Each role carries explicit `isUnique` and `isMandatory` constraint declarations.
4. Each entity type declares a `referenceScheme` identifying its preferred key.
5. Value types declare a `valueDataType` from the normative set (Text, Number, Date, Boolean, Currency).
6. Enumerated types export the complete set of allowed values as `enumValues`.
7. The serialization format preserves the predicate structure, semantic enrichment (alternate readings, sample values), and is machine-parseable (JSON).
8. Roles that produce foreign key columns with names differing from the referenced entity's reference scheme declare an explicit `roleName`.

Conformant implementations ensure that any automated system consuming the schema can determine the correct join semantics, cardinality, and participation constraints for every relation—replacing structural inference with a declared business grammar that is the single source of truth for text-to-SQL generation.

References

1. Codd, E. F. (1970). "A Relational Model of Data for Large Shared Data Banks." *Communications of the ACM*, 13(6), 377–387.
2. Halpin, T. (2006). *Information Modeling and Relational Databases*. 2nd ed. Morgan Kaufmann.
3. Date, C. J. (2003). *An Introduction to Database Systems*. 8th ed. Addison-Wesley.
4. Chen, P. P. (1976). "The Entity-Relationship Model—Toward a Unified View of Data." *ACM Transactions on Database Systems*, 1(1), 9–36.