Logical Data Model Naming Conventions

October 27, 2025 – Data-Only Scope

Pierre Bonnet – www.engage-meta.com

Purpose and Scope

This document defines the naming standards for the Logical Data Model (LDM). It translates the Business Data Model (BDM) into a LDM compliant with a further implementation in Supabase (PostgreSQL).

Table of contents

Position of the Logical Data Model

The Business Data Model (BDM) expresses business meaning and relationships without technical constraints. The Logical Data Model (LDM) refines it with precise structure, naming, and implementation rules consistent with SQL and Supabase. It serves as a stable bridge between business semantics and the physical database schema.

Naming Conventions

General Rules

Use lowercase snake_case for all identifiers. Keep names short and explicit, avoid reserved SQL words, and align with the Business Data Model. Prefer semantic clarity over technical prefixes.

Allowed characters:

- Identifiers (table, column, schema, constraint names) may contain lowercase letters (a–z), digits (0–9), and underscores (_) only.
- Avoid uppercase letters, spaces, hyphens, or special characters. PostgreSQL automatically lowercases unquoted identifiers, and Supabase enforces lowercase naming.

Maximum length:

- PostgreSQL limits identifiers to 63 bytes (≈ 63 characters in ASCII).
- To stay safe when generating indexes, constraints, and relationships automatically, keep all logical names ≤ 50 characters.

Invalid or risky characters:

- Do not use accented characters, symbols ($, -, /, .), or spaces.
- These may break migrations or Supabase API endpoints (PostgREST and GraphQL layers).

Schema and Table Naming Convention

Element	Convention	Example
Schema (Data Domain)	domain or domain_subdomain	admin_hr, finance_accounting, production
Table	singular noun; 'tb_' prefix	tb_employee_survey, tb_training_session_participation

Attributes Naming Convention

Attribute Type	Convention	Stereotype	Example
Surrogate Primary Key (UUID)	id_<table>	<<PK>>	id_tb_employee_survey <<PK>>
Business Primary Key	1 to N attributes used as a unique constraint to form the business primary key	<<BK>>	(code <<BK>> date <<BK>>)
Foreign Key	id_<referenced_table> In case of multiple foreign keys pointing to the same table: id_<referenced_table>_<association name> In case of a link to the generic Codification table: <id_tb_codification>_<cd_type>	<<FK>>	id_tb_employee <<FK>>
Code / Enum	cd_<meaning>		cd_state, cd_language
Date / Time	dt_<meaning> / ts_<meaning>		dt_created, ts_updated
Boolean	is_, has_, or can_ prefix		is_active, has_signed_contract

Views and Functions

Views: prefix 'vw_'; Functions: prefix 'fn_'.

Example: vw_employee_engagement_score, fn_calculate_training_hours.

Codification Management

Ensure consistent codes/labels, multilingual UX, and data quality without spawning dozens of tiny lookup tables while still giving first-class treatment to a few structural code lists (Country, Currency, Unit, Language…) used across the enterprise and/or tied to external standards (ISO/UN).

Use a Dedicated Reference Table (own table + FK) when the list is:

- Structural & cross-domain (used by many domains, part of keys, or drives joins).
- Externally standardized with rich attributes (e.g., ISO 3166-1 Country has alpha-2, alpha-3, numeric, region).
- Stable, low-churn (rarely changes; updates are governed).
- Needed in constraints (e.g., must be referenced as a strict FK—no "type" ambiguity).
- Examples: tb_country, tb_currency, tb_unit, tb_language.

Use the Generic Codification Table for lists that are:

- Functional / business-specific, evolving, or project-scoped.
- Medium/high churn (business adds/removes values).
- Mostly label-driven (labels, order, grouping, validity windows).
- Multilingual UX (labels resolved at runtime).
- Examples: EMPLOYMENT_STATUS, CONTRACT_TYPE, QUALITY_NONCONFORMITY_TYPE, ISO_CATEGORY, HR_LEVEL.

id_codification	cd_type	cd_value	cd_language	lb_short	lb_long	dt_validity_start	dt_validity_end
1	CONTRACT_TYPE	TEMP	EN	Temporary	Temporary employment contract	2020-01-01	*(null)*
2	CONTRACT_TYPE	PERM	EN	Permanent	Permanent employment contract	2020-01-01	*(null)*
3	EMPLOYEE_STATUS	ACTIVE	EN	Active	Currently employed and active	2020-01-01	*(null)*
4	EMPLOYEE_STATUS	LEAVE	EN	On Leave	Temporarily absent from work	2020-01-01	*(null)*
5	EMPLOYEE_STATUS	TERMINATED	EN	Terminated	Employment contract ended	2020-01-01	*(null)*
6	NONCONFORMITY_TYPE	PROD_DEFECT	EN	Product Defect	Defect found during production	2020-01-01	*(null)*

How Business Tables Use The tb_codification

Business tables store only the cd_value (e.g., 'FR') or the codification identifier. Labels are resolved dynamically using joins when needed for reporting or user interfaces.

Example join (conceptual):

```
JOIN common.codification c
  ON c.cd_type = 'COUNTRY'
 AND c.cd_value = employee.cd_country
```

Join the employee table with the codification table to translate the employee's stored country code (FR, VN…) into its full label ('France', 'Vietnam'…).

Historization

Historization refers to the ability to preserve the different versions of data over time, enabling audit, traceability, and 'as-of-date' analysis. This section describes three complementary strategies that can be used in Supabase/PostgreSQL depending on functional requirements, performance and storage considerations.

Option 1 – Separate "_history" Table (Classic Pattern)

This approach creates a dedicated table suffixed with '_history' for each entity that requires versioning. The main table always contains the latest (current) version, while the history table stores all previous versions with validity timestamps.

Main Table	History Table
employee_contract	employee_contract_history

Before the update

employee_contract

id_contract	id_employee	salary	updated_at
101	12	1 800	2024-01-01 09:00

employee_contract_history

id_contract	id_employee	salary	recorded_from	recorded_to
—	—	—	—	—

After the update

employee_contract

id_contract	id_employee	salary	updated_at
101	12	2 500	2025-01-01 10:00

employee_contract_history

id_contract	id_employee	salary	recorded_from	recorded_to
101	12	1 800	2024-01-01 09:00	2025-01-01 10:00

Advantages: Simple to understand, keeps main table small, easy ‘as-of’ reporting.

Drawbacks: Requires triggers to copy old rows; doubles the number of tables to maintain.

Option 2 – Bitemporal Validity Columns (Single Table)

In this approach, all versions are stored in the same table with two pairs of timestamp columns that describe both business validity and system recording time.

This enables advanced audit scenarios (‘what did we know on a given date’).

It is a structured historization based on two clocks (business + system).

Example:

employee_contract
| id_contract | id_employee | salary | ts_valid_from | ts_valid_to | ts_recorded_from |
ts_recorded_to |
|--------------|--------------|---------|----------------|---------------|----------------|
| 101 | 12 | 1 500 | 2023-01-01 | 2023-12-31 | 2023-01-01 | 2023-12-31 |
| 102 | 12 | 1 800 | 2024-01-01 | 9999-12-31 | 2024-01-01 | 9999-12-31 |

Column	Represents	Purpose	Example
ts_valid_from	Business validity start date	When the fact becomes true in real life (e.g., when a new contract or price takes effect).	The salary increase is valid starting January 1 2024.
ts_valid_to	Business validity end date	When the fact stops being true in real life (e.g., contract ends or price	The old salary stopped being valid on December 31 2023

		replaced).	
ts_recorded_from	System recording start time	When the information was stored or known by the system (technical time).	HR encoded the new contract on January 15 2024
ts_recorded_to	System recording end time	When the system replaced or deleted this version (i.e., when it was superseded by a newer row	This version remained in the DB until the next update on May 1 2024

Advantages: Full audit trail within one table; supports both business and system time (true bitemporal modeling). No need to maintain separate history tables.

Drawbacks: Heavier queries (must filter by validity); larger storage footprint; more complex to manage manually.

Option 3 – Insert-Only (Append-Only Pattern)

This strategy forbids updates entirely.

Each change inserts a new record instead of modifying existing data. Queries determine the latest valid record by filtering or sorting by timestamp. This approach is simple, auditable, and aligns well with modern event-driven architectures.

It is event log with everything is an append.

Example:

employee_salary

employee_id	salary	ts_valid_from	ts_valid_to	is_current
12	1 500	2023-01-01	2023-12-31	false
12	1 800	2024-01-01	9999-12-31	true

Advantages: No updates required; natural audit trail; compatible with append-only data pipelines. Partial unique indexes (e.g., one current row per key) can ensure data integrity – Example:

```
CREATE UNIQUE INDEX uq_contract_current
ON employee_contract (id_contract)
WHERE is_current = true;
```

Drawbacks: Table grows faster; small updates require inserting new rows; uniqueness constraints must be managed carefully.

Comparison with Insert-Only Pattern

Although both the Bitemporal and Insert-Only approaches rely on inserting new records instead of overwriting existing ones, they serve different purposes and offer different levels of temporal precision. The table below summarizes the key distinctions.

Aspect	Option 2 – Bitemporal Validity Columns	Option 3 – Insert-Only (Append-Only Pattern)
Purpose	Capture both business time (when facts are true in reality) and system time (when the database learns or replaces them).	Keep every change as a simple event for audit or analytics, without managing business validity.
Columns used	Four timestamps: ts_valid_from / ts_valid_to (business) + ts_recorded_from / ts_recorded_to (system).	One or two timestamps: ts_recorded or ts_valid_from, sometimes with an is_current flag.
Update behavior	Each change inserts a new row and closes the previous one (updates ts_recorded_to, optionally ts_valid_to).	Each change is a pure INSERT; no UPDATE at all. The latest record is found by the max timestamp or flag.
Query model	Can answer 'what was true in business at date X' and 'what did the system know at date Y'.	Can only reconstruct 'what was the latest record' or build event timelines.
Complexity	Higher– four timestamps, more logic in triggers and queries.	Lower – append-only logic, simpler to implement.
Use cases	Regulatory, compliance, or legal data requiring full traceability (contracts, financial transactions).	Event logs, telemetry, metrics, data lakes, analytical pipelines.
Integrity rules	Must avoid overlapping validity intervals; requires controlled updates.	Rely on uniqueness or partial indexes to identify current records.

In summary, every bitemporal table is append-only, but not every append-only table is bitemporal. Bitemporal modeling provides dual-time semantics (business and system validity), whereas Insert-Only simply records immutable events.

Recommended Practice

It is recommended to select the historization pattern per domain according to the data's business volatility and audit requirements:

- Use option 1 (History Table) for transactional domains (Finance, HR contracts, ISO).
- Use option 2 (Bitemporal) for regulatory or compliance-critical data where system and business validity both matter (e.g., Quality, ESG).
- Use option 3 (Insert-Only) for fast-changing operational data (IoT sensors, production metrics, logs).

In all cases, historization design must be documented in the Logical Data Model to guarantee long-term consistency, auditability, and performance predictability.

State

State management: use cd_state for current status; add multiple columns if parallel states exist.

Surrogate Business Primary Key (UUID)

Principle

Each business table in the Logical Data Model (LDM) must include a surrogate primary key implemented as a Universally Unique Identifier (UUID). This ensures global uniqueness of records across environments, projects, and future integrations, regardless of the physical database instance or deployment context:

1. Durability and portability. UUIDs remain stable across database migrations, data exchanges, and system merges.
2. Avoids collisions. No dependency on incremental sequences that can overlap between environments (e.g., dev, staging, prod).
3. Aligns with distributed architectures. Supabase, like most modern cloud and event-driven platforms, natively supports UUIDs as uuid data type with default generator gen_random_uuid() or uuid_generate_v4().
4. Future interoperability. Guarantees continuity with potential external systems (ERP, AI services, API-driven ecosystems).

Implementation Guidelines

Element	Rule
Primary key column name	Always id_<table_name> (e.g., id_tb_employee, id_tb_invoice).
Data type	uuid
Default value	gen_random_uuid() (requires PostgreSQL pgcrypto extension; enabled by default in Supabase).
Uniqueness	Defined as the primary key of the table.
Business key	Optionally, define a natural or business key (e.g., cd_employee, cd_invoice) to maintain readability and traceability.
Referencing foreign keys	Use UUID type for all foreign keys referencing surrogate PKs to ensure type consistency.

Example

Example implementation in Supabase/PostgreSQL:

```sql
CREATE TABLE admin_hr.employee (
    id_employee uuid PRIMARY KEY DEFAULT gen_random_uuid(),
    cd_employee text UNIQUE NOT NULL,
    full_name text NOT NULL,
    hire_date date NOT NULL
);
```

Best Practices

- Always generate UUIDs server-side to guarantee integrity and uniqueness.
- Keep business codes (cd_*) human-readable for reporting, but never rely on them as technical identifiers.
- Avoid mixing integer sequences and UUIDs in the same logical model. Choose one consistent approach for all entities.
- When importing legacy data, retain the historical business key as cd_legacy or cd_external for traceability.

Logical Model Structuration in Visual Paradigm

Organize the LDM in Visual Paradigm (VP) to mirror the Business Data Model (BDM) ensuring traceability, clarity, and a seamless transition to the physical schema (pgModeler).

BDM: Class Diagram in VP.

LDM: ERD Diagram in VP.

Maintain Package Continuity with the BDM

- Data Domain → VP package (e.g., HR, Finance, Production).
- Data Sub-Domain → sub-package (e.g., HR.Survey, HR.Training).
- One logical diagram per sub-package unless it contains very few tables.

--- end---