PLAN 模型结构定义

1. 文档定位与作用 (Document Scope & Purpose)

1.1 文档定位

本文档是 NL2SQL 系统的 **核心数据契约 (Data Contract)** 定义文档。
它定义了系统内部流转的核心数据结构 —— **PLAN (Query Plan)**。该结构承载了从“自然语言意图”到“数据库可执行逻辑”的中间态表达。

> **预期读者**：前端开发（了解返回结构）、LLM Prompt 工程师（了解生成目标）、测试人员（构造测试用例）、后端开发（遵循数据协议）。

1.2 适用范围

PLAN 对象贯穿整个流水线的核心链路，作为各阶段的**标准输入/输出契约**：

- **Stage 2 (Output)**: LLM 生成的初步意图（Skeleton Plan）。
- **Stage 3 (Input/Output)**: 校验与补全逻辑的操作对象。
- **Stage 4 (Input)**: SQL 生成器的标准输入（Validated Plan）。
- **Stage 6 (Input)**: 结果解释时的上下文依据。

1.3 文档生态关系 (Document Ecosystem)

PLAN 模型并非孤立存在，它必须与以下文档配合使用：

- **上游依赖：《语义层配置规范》 (Source of IDs)**
  - PLAN 中出现的所有业务标识符（如 METRIC_GMV），必须严格引用该文档中的定义。**严禁使用物理表名或临时编造的字段名。**
- **下游实现：《详细设计文档 (LLD)》 (Implementation Logic)**
  - **本文档 (Schema)**：定义 “数据长什么样”（如：limit 允许为 null）。
  - **LLD 文档 (Logic)**：定义 “数据怎么处理”（如：若 limit 为 null，Stage 3 代码将其补全为 100）。

2. 设计前提与约束（Design Prerequisites & Constraints）

PLAN 模型的设计并非凭空构建，而是严格依赖于以下两大基础地基。任何对 PLAN 结构的修改或扩展，都必须首先检视是否符合这两大前提，并严格遵守随之而来的约束。

2.1 设计前提 (Design Prerequisites)

PLAN 的结构形态是由**业务边界**和**语义依赖**共同决定的。

1. 地基一：有限的业务分析边界 (Bounded Analytical Scope)
PLAN 的设计初衷**不是**为了覆盖 SQL 的所有语法特性（如复杂的嵌套、递归、存储过程），而是为了精准覆盖 MVP 阶段的核心 BI 分析需求(包含：求和（Sum）、计数（Count）、分组（Group By）、同环比（YoY）、筛选（Filter）、排序（Top N））。

- **前提说明**：系统仅支持“聚合”、“趋势”、“明细”三类原子意图。
- **对 PLAN 的影响**：
  - intent 字段被严格锁定为枚举值（AGG/TREND/DETAIL）。
  - 不支持多层嵌套结构，因为复杂问题已被上游拆解为多个独立的原子 PLAN。

2. 地基二：强语义层依赖 (Strong Semantic Dependency)
PLAN 是“自然语言”与“物理数据”之间的**语义中间态**。它不包含任何物理数据库的信息（如表名、字段类型），而是完全依赖外部的**语义配置**来解释其含义。

- **前提说明**：PLAN 本身不存储元数据，它只是一个承载“语义标识符（ID）”的容器。
- **对 PLAN 的影响**：
  - PLAN 中出现的所有 ID，必须能在 SemanticRegistry（语义注册表）中找到定义。
  - PLAN 不关心“怎么查”（Join 路径），只关心“查什么”（语义对象）。

2.2 核心约束 (Core Constraints)

2.1.1 结构性约束 (Structural Constraints)

本节定义 PLAN 对象在数据结构（JSON Shape）上必须遵守的物理定律。

1. 扁平化约束 (Flatness Constraint)

- PLAN 结构**不支持递归或嵌套**。
- JSON 中严禁出现 sub_plans 或 children 等嵌套字段。复杂的嵌套逻辑必须由上游（Stage 1）拆解为多步查询。

2. 声明式约束 (Declarative Constraint)

- PLAN 只描述 **“要什么 (What)”**，不描述 **“怎么取 (How)”**。
- JSON 中**严禁**出现物理 SQL 概念，如 join_type (Left/Inner)、table_name、primary_key。

3. 无状态约束 (Stateless Constraint)

- 每个 PLAN 对象必须是**自包含 (Self-contained)** 的。
- 后端生成 SQL 时，仅依赖当前 PLAN 内容，不依赖上一次交互的上下文缓存。

2.1.2 语义完整性约束 (Semantic Integrity Constraints)

本节定义 PLAN 对象内部的值（Values）必须遵守的数据质量标准。

1. 接口层标识符规范 (Interface Identifier Conventions)
为了在接口层面明确区分语义对象，PLAN 中引用的 ID 必须严格遵循以下前缀：

- **指标 (Metric)**
  - 前缀：METRIC_ (如 METRIC_GMV)
  - 约束：必须在 metrics 或 filters 列表中使用。
- **维度 (Dimension)**
  - 前缀：DIM_ (如 DIM_REGION)
  - 约束：必须在 dimensions 或 filters 列表中使用。
- **逻辑过滤器 (Logical Filter)**
  - 前缀：LF_ (如 LF_VALID_ORDER)
  - 约束：仅限在 filters 列表中作为引用 ID 使用。

> **排除项说明 (Exclusions)**：
>
> **实体 (Entity) / 时间窗口 (Time Window) / 权限角色 (Role)**：属于后端推导逻辑或隐式上下文，**不属于 PLAN 接口协议的一部分**，因此不在此定义命名规范，全量命名规范详见《语义层配置规范》）。

2. 存在性校验 (Existence Verification)

- PLAN 中出现的所有 ID，必须能在系统启动时加载的 SemanticRegistry（语义注册表）中查找到对应定义。
- **严禁**使用注册表中不存在的“幻觉 ID”。

3. 严格匹配原则 (Strict Matching)

- LLM 输出的 ID 必须与 YAML 定义 **完全一致**（区分大小写，通常全大写）。
- 任何拼写错误（如 METRIC_gmv 或 metric_GMV）都将被视为非法，并在 Stage 3 校验中被剔除。

3. PLAN结构

3.1 顶层结构概览 (Structure Overview)

以下展示了一个包含所有核心特性的完整 PLAN 示例。
场景假设：查询“华北区”最近30天按日统计的销售额同比趋势，且只看自营渠道，按日期升序排列。

```json
{
  "intent": "TREND",
  "metrics": [
    {
      "id": "METRIC_GMV",
      "compare_mode": "YOY"
    }
  ],
  "dimensions": [
    {
      "id": "DIM_ORDER_DATE",
      "time_grain": "DAY"
    }
  ],
  "filters": [
    {
      "id": "DIM_REGION",
      "op": "EQ",
      "values": ["North China"]
    },
    {
      "id": "DIM_CHANNEL",
      "op": "IN",
      "values": ["Self-Operated"]
    }
  ],
  "time_range": {
    "type": "LAST_N",
    "value": 30,
    "unit": "DAY",
    "start": null,
    "end": null
  },
  "order_by": [
    {
      "id": "DIM_ORDER_DATE",
      "direction": "ASC"
    }
  ],
  "limit": 100,
  "warnings": []
}
```

3.2 字段详细定义 (Checklist)

以下是**最终定稿**的 PLAN 字段详细定义：

1. intent

- **类型**：String
- **是否必填**：**是**
- **说明**：用户查询的核心意图，直接决定 SQL 生成的主模板。
- **取值约束**：仅限以下 3 个枚举值：
  - "AGG" (聚合统计)：例如“计算总销售额”、“按地区统计人数”。
  - "TREND" (趋势分析)：例如“最近 30 天的销售趋势”。
  - "DETAIL" (明细查询)：例如“列出最近的 10 个订单”。
- **科学依据**：
  - **SQL 映射唯一性**：这三个意图分别对应 SQL 的三种根本形态（GROUP BY 聚合、DATE_TRUNC 时间轴聚合、无聚合 SELECT *），涵盖了 MVP 阶段 95% 的分析需求。
- **前置依赖**：
  - 无。这是系统硬编码的顶层逻辑分支。

2. metrics

- **类型**：List
- **是否必填**：否 (当 intent=DETAIL 时可为空)
- **说明**：查询涉及的指标列表。
- **元素结构**：{ "id": "METRIC_ID", "compare_mode": "YOY" }
- **约束**：
  - id: 必须是 semantic_metrics.yaml 中定义的 metric_id (如 METRIC_GMV)。
  - compare_mode: 仅限以下枚举值或 null：
    - null (默认，无对比)
    - "YOY" (Year-Over-Year，同比)
    - "MOM" (Month-Over-Month，环比)
    - "WOW" (Week-Over-Week，周环比)
- **科学依据**：
  - **计算可达性**：YOY/MOM/WOW 是标准的时间偏移计算，后端可以通过 SQL 的 JOIN 自身或窗口函数 (LAG) 稳定实现。

    - **复杂度控制**：排除自定义偏移（如“向前推 3 天”），避免 MVP 阶段陷入复杂的参数化计算泥潭。
- **前置依赖**：
    - **YAML 配置**：semantic_metrics.yaml 必须存在，且 ID 必须唯一。

3. dimensions

- **类型**：List
- **是否必填**：否
- **说明**：查询涉及的维度列表 (即 SQL GROUP BY 的对象)。
- **元素结构**：{ "id": "DIM_ID", "time_grain": "MONTH" }
- **约束**：
    - id: 必须是 semantic_core.yaml 中定义的 dimension_id (如 DIM_COUNTRY)。
    - time_grain: 仅限以下枚举值或 null：
        - null (非时间维度，或不强制粒度)
        - "DAY" (按天)
        - "WEEK" (按周)
        - "MONTH" (按月)
        - "QUARTER" (按季)
        - "YEAR" (按年)
- **科学依据**：
    - **数据库兼容性**：这 5 个粒度是所有主流数据库 (MySQL, PG, ClickHouse) 原生函数 (DATE_TRUNC, DATE_FORMAT) 都支持的标准集合。
- **前置依赖**：
    - **YAML 配置**：semantic_core.yaml 中对应的时间维度必须配置 allowed_time_grains，后端需校验 LLM 输出的粒度是否在允许列表中。

4. filters

- **类型**：List
- **是否必填**：否
- **说明**：过滤条件集合。LLM 不区分 WHERE/HAVING，统一输出，由后端根据 ID 类型自动分流。
- **元素结构**：{ "id": "TERM_ID", "op": "EQ", "values": [...] }
- **约束**：

    - id: 必须是合法的 Metric ID 或 Dimension ID。
    - op: 仅限以下枚举值：
        - "EQ" (等于 =)
        - "NEQ" (不等于 != / <>)
        - "IN" (包含于 IN (...))
        - "NOT_IN" (不包含于 NOT IN (...))
        - "GT" (大于 >)
        - "LT" (小于 <)
        - "GTE" (大于等于 >=)
        - "LTE" (小于等于 <=)
        - "BETWEEN" (区间 BETWEEN a AND b)
        - "LIKE" (模糊匹配 LIKE '%...%')
    - values: 必须是 **List (JSON Array)**。元素必须为**非空 (Non-null)** 的原始类型 (String/Number/Boolean)，禁止后端做隐式类型转换。
- **科学依据**：
    - **ANSI SQL 完备集**：上述操作符构成了 SQL 过滤能力的最小完备集。
    - **类型安全**：强制要求 values 为 List 且保持原类型，是为了配合后端 SQL 生成器使用“参数化查询” (Parameter Binding)，防止 SQL 注入。
- **前置依赖**：
    - 无。

5. time_range

- **类型**：Object
- **是否必填**：否 (若用户未提及时间，必须为 null)
- **说明**：全局时间范围限制。
- **结构**：{ "type": "...", "value": ..., "unit": "...", "start": "...", "end": "..." }
- **约束**：
    - type: 仅限 "LAST_N" (相对时间) 或 "ABSOLUTE" (绝对区间)。
    - unit: 仅限 "DAY", "WEEK", "MONTH", "QUARTER", "YEAR"。
    - value: 正整数。
    - start / end: 必须符合 **ISO 8601 日期格式 (YYYY-MM-DD)**。
- **科学依据**：

- **场景覆盖**：LAST_N 解决动态看板需求（每天看都是最新的），ABSOLUTE 解决固定报告需求（复盘历史）。
- **计算解耦**：将“相对时间计算”（如 NOW() - 7 days）交给后端或数据库处理，LLM 只负责提取数字。

- **前置依赖**：
  - **默认值配置**：后依赖 semantic_metrics.yaml 中的 default_time 配置（作为 Stage 3 补全逻辑的数据源）。

6. order_by

- **类型**：List
- **是否必填**：否
- **说明**：排序规则。
- **元素结构**：{ "id": "TERM_ID", "direction": "DESC" }
- **约束**：
  - id: 必须出现在同级 PLAN 的 metrics 或 dimensions 列表中 (即只能对已查询的列排序)。
  - direction: 仅限 "ASC" (升序) 或 "DESC" (降序)。
- **科学依据**：
  - **SQL 语法限制**：标准 SQL 通常要求 ORDER BY 的字段必须在 SELECT 或 GROUP BY 中有效。
- **前置依赖**：
  - 无。

7. limit

- **类型**：Integer
- **是否必填**：否
- **说明**：结果集行数限制。
- **约束**：
  - 必须是正整数 (>0)。
  - 若 LLM 未输出，后端必须强制注入默认值 (如 100)。
- **科学依据**：
  - **系统保护**：防止用户无意中查询全表数据导致内存溢出 (OOM)。
- **前置依赖**：
  - 后端配置中需定义 DEFAULT_LIMIT 和 MAX_LIMIT 常量。

8. warnings

- **类型**：List[String]
- **是否必填**：否 (默认为空列表 [])
- **说明**：解析与校验过程中的告警信息列表。用于记录“非致命错误”的处理结果，供后续阶段（如 Answer 生成）向用户解释查询结果的偏差。
- **元素结构**：简单的描述性字符串。例如："Ignored unknown metric ID: METRIC_PROFIT"。
- **约束**：
  - **生成源**：严禁 LLM 生成此字段。必须由 **后端解析代码 (Stage 2 Parser)** 或 **校验逻辑 (Stage 3)** 在执行清洗操作（如剔除幻觉 ID、修正非法参数）时自动追加。
  - **生命周期**：
    i. **Stage 2 (解析层)**：初始化为空列表。若发现幻觉 ID 并执行剔除，在此追加告警（如 "已忽略未知指标: METRIC_XYZ"）。
    ii. **Stage 3 (校验层)**：若触发业务降级或默认值填充，在此追加告警（如 "未指定时间，已默认最近30天"）。
  - **内容**：必须包含被丢弃/修改的对象的原始名称或 ID，以便追溯。
- **科学依据**：
  - **系统透明度 (Transparency)**：避免“用户问 A 和 B，系统只查了 A 却不告诉用户 B 不存在”的糟糕体验。
  - **优雅降级 (Graceful Degradation)**：允许 Plan 在存在部分错误（如 5 个指标中有 1 个幻觉）的情况下继续执行，而不是直接报错阻断，同时保留了错误现场。
- **前置依赖**：
  - 后端解析逻辑必须支持“容错模式”：遇到非法 ID 时不抛出异常，而是记录 Warning 并继续处理。

架构决策说明 (Architecture Decisions / FAQ)

本章解释“为什么这么设计”，防止后人推翻正确的设计。

4.1 为什么采用扁平化设计？（对比嵌套式 SQL 的优势）

4.2 为什么要删除 entities 字段？（解释“推导优于显式定义”的逻辑）

4.3 为什么不支持复杂的自定义时间偏移？（MVP 聚焦原则）

4. 架构决策说明 (Architecture Decisions / FAQ)

本章记录 PLAN 模型设计的关键架构决策（ADR）。这些决策是在权衡了 **LLM 能力边界**、**工程复杂度** 与 **业务覆盖率** 后的最优解。

4.1 为什么采用扁平化设计？(Why Flat Design?)

决策

PLAN 结构不支持嵌套（Nested）或递归，强制保持单层扁平结构。

背景

传统 NL2SQL 方案常让模型生成类似 AST（抽象语法树）的嵌套 JSON，以表达“查询销售额大于平均值的城市”这类复杂逻辑。

决策理由

1. **LLM 稳定性**：实验表明，JSON 嵌套越深，LLM 语法错误率呈指数级上升。扁平结构类似“填表”，对 LLM 最友好。
2. **职责分离**：
   - **复杂逻辑**（如嵌套子查询）由 **Stage 1** 拆解为原子问题。
   - **原子映射** 由 **Stage 2** 处理。PLAN 只需承载原子意图。
3. **解析确定性**：扁平结构可直接映射 Pydantic 模型，无需编写复杂的递归解析器，维护成本极低。

> **核心哲学**：本设计本质是用 **“后端代码的确定性”** 去对冲 **“LLM 的不确定性”**。

4.2 为什么要删除 entities 字段？(Why Remove 'entities'?)

决策

PLAN 不再要求 LLM 输出 `entity_id`，改为由后端根据 `metrics` 和 `dimensions` 自动推导。

背景

早期设计中包含 `entities` 字段，要求 LLM 显式判断“这是查订单表还是查用户表”。

决策理由（权衡分析）

- **一致性（删除胜 ✅）**：若保留，LLM 可能输出“销售指标”配“人事实体”，导致后端冲突。删除后，Entity 由 Metric 唯一推导，**数学上绝无冲突**。
- **Token 消耗（删除胜 ✅）**：实体信息已隐含在指标中，删除可节省 Token。
- **Prompt 复杂度（删除胜 ✅）**：LLM 只需关注“查什么指标”，无需理解实体概念，认知负担低。
- **纯筛选场景（保留胜 ⚠）**：无指标时（如“看北京的数据”），删除该字段会导致后端无法定位主表。

最终结论

利远大于弊。针对“纯筛选场景”的劣势，MVP 阶段采用 **“默认主表策略”**（如出现 `DIM_CITY` 默认关联销售表）解决。

4.3 为什么不支持复杂的自定义时间偏移？(Why No Custom Time Offsets?)

决策

`compare_mode` 仅支持枚举值（YOY/MOM），不支持“向前推 N 天”或“对比任意时间段”。

背景

用户常有非标准对比需求，如“和上周二对比”、“和 3.5 天前对比”。

决策理由

1. **MVP 聚焦 (Pareto Principle)**：BI 场景中 **95%** 的需求集中在同比和环比。
2. **工程稳定性**：任意时间偏移涉及复杂的 Window Function 或 Self-Join，且自然语言解析（如“上周二”）极具不确定性。
3. **策略**：我们在 Stage 1 拆解或 Stage 2 明确拒绝此类需求，以换取核心链路的**绝对稳定**。

4.4 为什么意图只有3类，没有 `COMPARE`？

在 NL2SQL 的架构演进中，关于 `COMPARE`（对比）到底应该是一个**“独立意图（Intent）”**还是一个“修饰能力（Capability）”，**一直有两派观点。我们选择的是**后者**（即修饰能力），这是更先进、更灵活的设计。

我们定义的 `intent`（AGG/TREND/DETAIL），本质上是在定义 **SQL 结果集的形状（Shape of Result）**：

- **AGG**: 返回一行或几行汇总数据（标量/字典）。
- **TREND**: 返回一个时间序列（X轴是时间，Y轴是值）。
- **DETAIL**: 返回一张二维明细表。

那么，`COMPARE` 是什么形状?

- 如果是“今年 vs 去年”的对比：它的形状依然是 **AGG**（两个数字放在一起）。
- 如果是“今年每月 vs 去年每月”的趋势对比：它的形状依然是 **TREND**（两条曲线）。

结论：

如果你把 `COMPARE` 设为第 4 个意图，LLM 就会陷入**分类困境**：

> *用户问：“看下今年和去年的月度销售额趋势对比。”*
> *LLM 崩溃了：这到底是 `TREND`（因为有趋势）？还是 `COMPARE`（因为有对比）？*

为了避免这种歧义，我们将 `COMPARE` 降级为 **指标的一个属性**，而不是**整个查询的意图**。

并且：我们没有丢掉功能，而是把它拆解到了两个地方：

场景 A：时间对比（同比/环比）

- **用户问**：“销售额同比增长了多少？”
- **传统方案 (Intent=COMPARE)**：后端需要专门写一套处理 COMPARE 的逻辑。
- **我们的方案 (Intent=AGG + compare_mode)**：
- codeJSON

```
代码块
1  {
2    "intent": "AGG",
3    "metrics": [
4      { "id": "METRIC_GMV", "compare_mode": "YOY" } // <--- 在这里处理
5    ]
6  }
```

  - **优势**：这种结构允许**“混合查询”**。比如“我看销售额（不对比）和订单数（要同比）”，我们的结构能表达，但 `Intent=COMPARE` 的结构很难表达混合逻辑。

场景 B：维度对比（A vs B）

- **用户问**：“对比一下北京和上海的销售额。”
- **传统方案 (Intent=COMPARE)**：后端需要特殊处理。
- **我们的方案 (Intent=AGG + Filter)**：
  这本质上就是一个**带筛选的分组统计**，完全不需要特殊意图。
- codeJSON

```
代码块
1  {
2    "intent": "AGG",
3    "dimensions": [{ "id": "DIM_CITY" }], // 按城市分组"filters": [
4      { "id": "DIM_CITY", "op": "IN", "values": ["北京", "上海"] } // 只选这两个
5    ],
6    "metrics": [{ "id": "METRIC_GMV" }]
7  }
```

  - **优势**：后端 SQL 生成器不需要写任何额外代码，标准的 `GROUP BY` + `WHERE` 就能解决。