**Design Category system**

**场景**

* **设计一个 商品分类展示系统（Catalog / Category System），用户通过网页浏览商品。**
* **商品可以属于 多个分类（多对多关系）。**
* **分类可以有 多级层次（父分类、子分类），但前端每次只需要展示 直属子类。**

**用户需求**

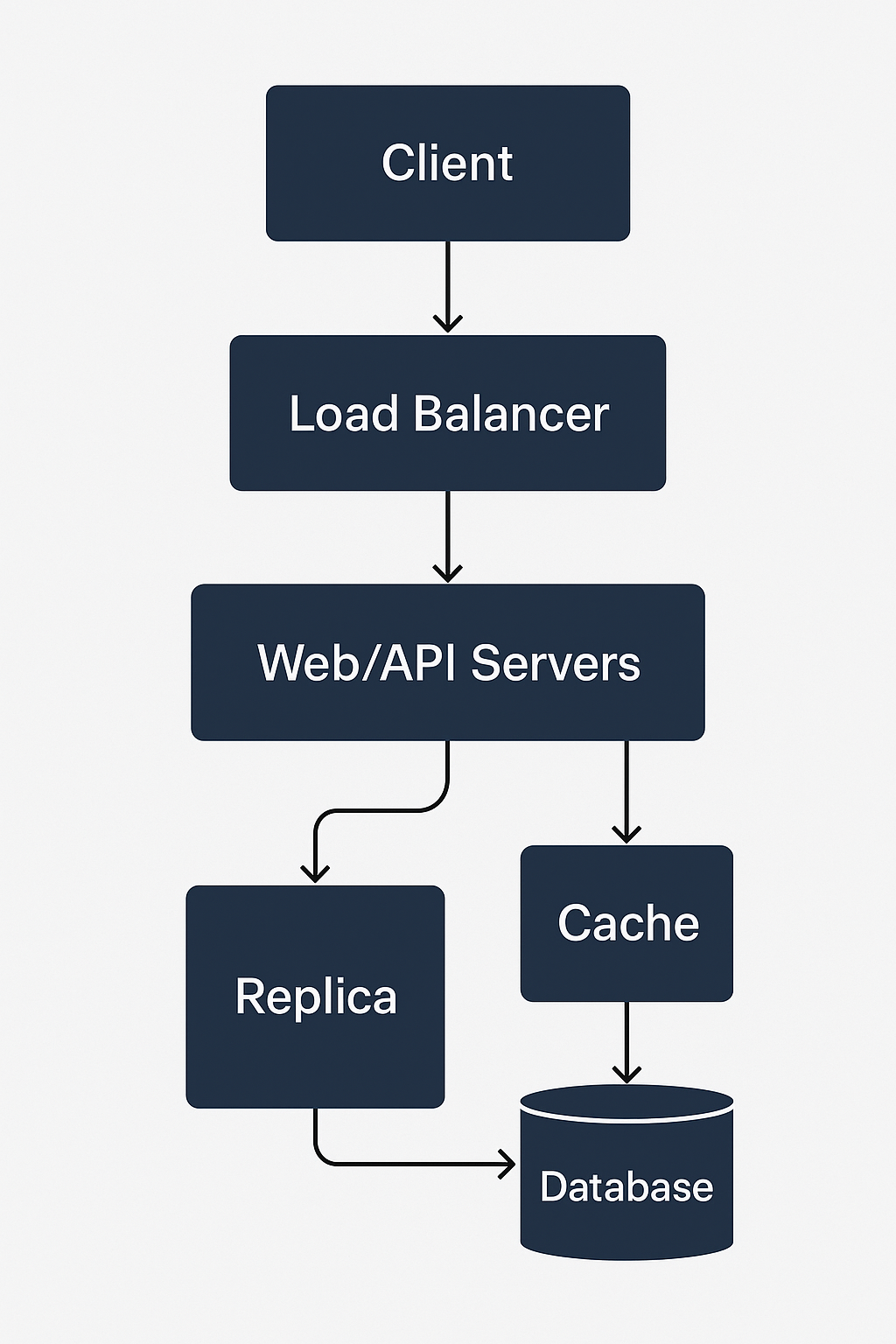
1. **用户打开网页时：**
   * **显示所有顶层分类（例如 Food、Clothes）。**
   * **显示部分商品（热门商品或顶层分类商品）。**
2. **用户点击某个分类：**
   * **显示该分类的 直属子分类。**
   * **显示该分类下的 商品。**
   * **不需要递归显示所有子孙分类（简化查询和缓存）。**
3. **商品可能属于 多个分类：**
   * **一个商品可能在不同分类页面同时出现。**

**面试考察点**

1. **数据库设计**
   * **如何存储分类（Category 表自关联实现树形结构）。**
   * **如何存储商品与分类的多对多关系（Product / ProductCategory 表）。**
2. **API 设计**
   * **如何设计请求与返回格式。**
   * **如何让前端方便地 fetch 分类和商品信息。**
3. **数据访问模式**
   * **如何查询顶层分类。**
   * **如何查询某分类下的直属子分类和商品。**
   * **如何处理商品属于多个分类的情况。**
4. **系统扩展性 / 高可用性（follow-up）**
   * **如何缓存分类和商品信息。**
   * **如何在大量用户访问时扩展系统。**
   * **如何保证系统稳定运行。**

**重点注意**

* **前半部分面试会偏重 API + 数据库 schema。**
* **后半部分可能会问 整体系统架构、Cache、Load Balancer、数据同步策略 等细节。**
* **面试中需要清楚表达：**
  1. **数据在数据库中的存储形式。**
  2. **请求格式和响应格式。**
  3. **前端 fetch 的逻辑。**



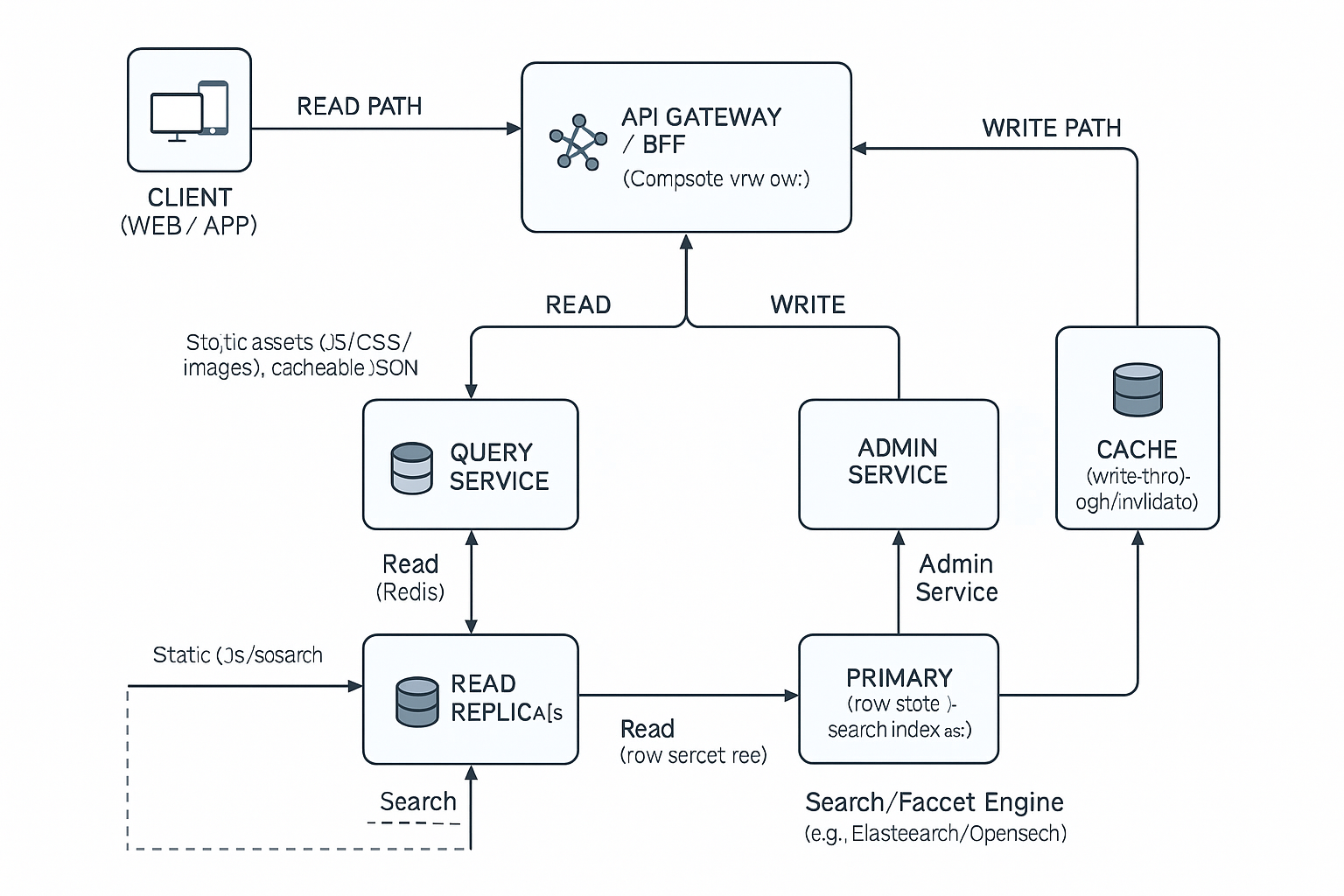
**题目核心**

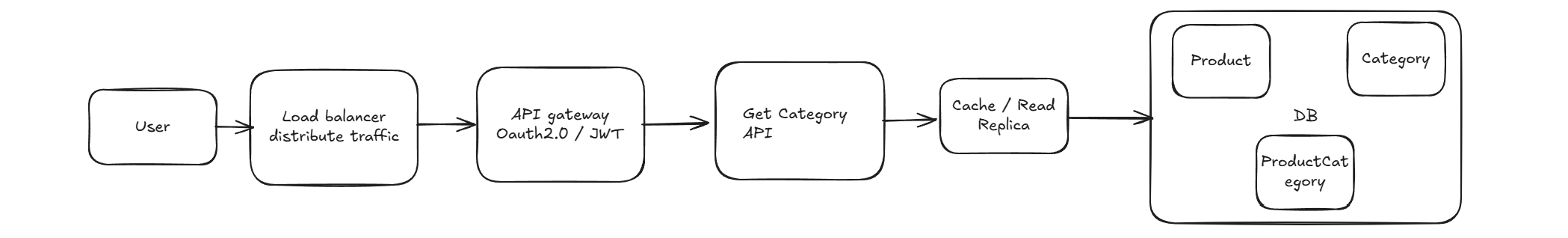
**目标：设计一个展示商品（Product）的分类（Category）系统的网页。**

* **用户打开页面时，可以看到所有顶层 Category（例如 Food、Clothes）。**
* **点击某个 Category 时，可以看到其直属子类（Sub-category）和该类下的商品。**
* **每个商品可能属于多个 Category。**
* **系统需要支持频繁展示和查询，并能扩展到大规模用户。**

**面试重点：**

1. **数据库 schema 设计（self-join Category 表 + Product 表 + 多对多关系）**
2. **API 设计（清晰描述 request / response）**
3. **数据访问模式（前端如何 fetch categories 和商品信息）**
4. **可扩展性 / cache / scaling / high availability**

****

****

**用户请求先经过 Load Balancer，再经过api gateway的token验证分发到多台 Web/API Server；读请求：优先从 Replica 或 Cache 命中，降低数据库压力；扩展性：所有 Web/API Server 都是无状态的，可以横向扩展；Cache、Replica、DB 也能独立扩展和集群化**

**数据库 Schema**

**2.1 Category 表**

**CREATE TABLE Category (**

**id BIGINT PRIMARY KEY,**

**name VARCHAR(255) NOT NULL,**

**parent\_id BIGINT NULL,**

**created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,**

**updated\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,**

**FOREIGN KEY (parent\_id) REFERENCES Category(id)**

**);**

**特点：**

* **典型做法是 自关联（Self-join），实现树形结构**
* **每个 Category 可以有父 Category（parent\_id），顶层 Category 的 parent\_id 为 null**
* **self-join 支持多级分类**
* **查询某个 category 的直属子类：**

**SELECT \* FROM Category WHERE parent\_id = :id;**

**2.2 Product 表**

**CREATE TABLE Product (**

**id BIGINT PRIMARY KEY,**

**name VARCHAR(255) NOT NULL,**

**description TEXT,**

**price DECIMAL(10,2),**

**image\_url VARCHAR(255),**

**created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,**

**updated\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP**

**);**

**2.3 ProductCategory 表（多对多关系）**

* **一个商品可能属于多个 Category**
* **一个 Category 也可能包含多个商品**

**CREATE TABLE ProductCategory (**

**product\_id BIGINT NOT NULL,**

**category\_id BIGINT NOT NULL,**

**PRIMARY KEY (product\_id, category\_id),**

**FOREIGN KEY (product\_id) REFERENCES Product(id),**

**FOREIGN KEY (category\_id) REFERENCES Category(id)**

**);**

**2.4 数据访问模式（Query Patterns）**

1. **获取顶层 Category：**

**SELECT \* FROM Category WHERE parent\_id IS NULL;**

1. **获取某 Category 的直属子类：**

**SELECT \* FROM Category WHERE parent\_id = :category\_id;**

1. **获取某 Category 下的商品（只直属商品）：**

**SELECT p.\***

**FROM Product p**

**JOIN ProductCategory pc ON p.id = pc.product\_id**

**WHERE pc.category\_id = :category\_id;**

**3️⃣ API 设计**

**3.1 获取顶层 Category**

**Request**

**GET /api/categories?depth=1&include=children,products**

**Query 参数**

* **depth：整数，预取层数上限（含当前层）。建议 ≤3，默认 1。**
* **include：children、products，可逗号组合；默认 children,products。**
* **products\_mode：none|summary|full，默认 summary（深层仅返回精简字段）。**
* **sort：popularity\_desc | price\_asc | price\_desc | newest（可选）。**
* **limit / offset：控制商品分页（可选）。**

**Response**

**[**

**{ "id": 1, "name": "Food", "parent\_id": null },**

**{ "id": 2, "name": "Clothes", "parent\_id": null }**

**]**

**3.2 获取某 Category 下的子类和商品（支持 depth）**

**Request**

**GET /api/categories/:id?depth=2&include=children,products&products\_mode=summary**

**Response**

**{**

**"category": { "id": 1, "name": "Food", "parent\_id": null },**

**"subcategories": [**

**{ "id": 3, "name": "Fruits", "parent\_id": 1 },**

**{ "id": 4, "name": "Vegetables", "parent\_id": 1 }**

**],**

**"products": [**

**{ "id": 101, "name": "Apple", "price": 1.5, "image\_url": "..." },**

**{ "id": 102, "name": "Banana", "price": 0.8, "image\_url": "..." }**

**],**

**"tree": {**

**"id": 1,**

**"children": [**

**{**

**"id": 3,**

**"name": "Fruits",**

**"products\_summary": [**

**{ "id": 201, "name": "Orange", "price": 2.0, "image\_url": "..." },**

**{ "id": 202, "name": "Lemon", "price": 1.8, "image\_url": "..." }**

**],**

**"children": []**

**},**

**{ "id": 4, "name": "Vegetables", "products\_summary": [], "children": [] }**

**]**

**},**

**"paging": { "page": 1, "limit": 20, "total": 187, "next\_page": 2 },**

**"meta": { "sort": "popularity\_desc", "cache\_ttl\_sec": 600 }**

**}**

**3.3 获取某 Category 的商品列表（完整分页）**

**Request**

**GET /api/categories/:id/products?limit=20&offset=0&sort=price\_asc**

**Response**

**{**

**"data": [**

**{ "id": 101, "name": "Apple", "price": 1.5, "image\_url": "..." },**

**{ "id": 105, "name": "Cherry", "price": 3.6, "image\_url": "..." }**

**],**

**"paging": { "limit": 20, "offset": 0, "total": 187, "next\_offset": 20 },**

**"meta": { "sort": "price\_asc" }**

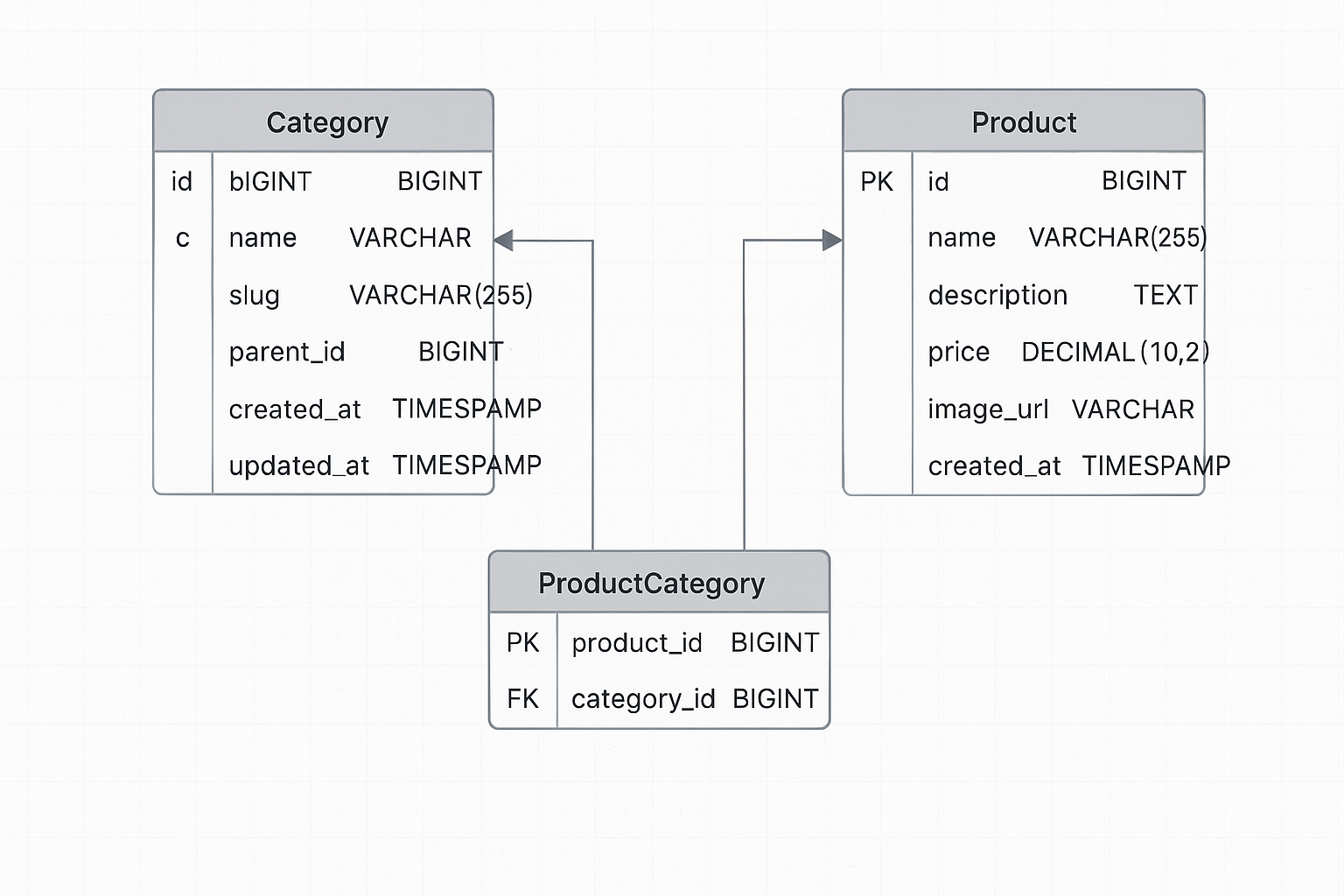
**}**

**数据库 Schema（核心三表）**

**特点：**

* **典型做法是 自关联（Self-join），实现树形结构**
* **每个 Category 可以有父 Category（parent\_id），顶层 Category 的 parent\_id 为 null**
* **self-join 支持多级分类**
* **查询某个 category 的直属子类：**

**SELECT \* FROM Category WHERE parent\_id = :id;**



**为什么 self-join 适合无论是有限层级还是无限层级的分类系统:**

In my design, I use a self-join table for categories, because it works really well whether the hierarchy is bounded or unbounded. Each category just points to its parent, so the structure is simple and flexible. If the business only needs three levels today, that’s fine—we just stop at three. But if tomorrow we need a fourth or even more, we don’t have to change the schema, we just insert more rows. For queries, getting direct children is always a single WHERE parent\_id = ? query, which is really efficient. So the main advantages are clarity, flexibility, and future-proofing. It keeps the schema simple while leaving room for growth.

**CREATE TABLE Category (**

**id BIGINT PRIMARY KEY,**

**name VARCHAR(255) NOT NULL,**

**parent\_id BIGINT NULL,**

**FOREIGN KEY (parent\_id) REFERENCES Category(id) // 一条分类记录的 parent\_id 可以等于另一条分类记录的 id，从而表示“我是那个分类的子类” 如果 parent\_id = NULL，就说明这个分类没有父类，是顶层分类。A category record’s parent\_id can equal the id of another category record, which means ‘I am a subcategory of that category.’ If parent\_id is NULL, it indicates that the category has no parent and is a top-level category.**

**);**

**CREATE TABLE Product (**

**id BIGINT PRIMARY KEY,**

**name VARCHAR(255) NOT NULL,**

**description TEXT,**

**price DECIMAL(10,2),**

**image\_url VARCHAR(255)**

**);**

**CREATE TABLE ProductCategory (**

**product\_id BIGINT NOT NULL,**

**category\_id BIGINT NOT NULL,**

**PRIMARY KEY (product\_id, category\_id),**

**FOREIGN KEY (product\_id) REFERENCES Product(id),**

**FOREIGN KEY (category\_id) REFERENCES Category(id)**

**);**

* **Category：自关联，支持无限层级，也适用于有限层级。**
* **Product：商品信息。**
* **ProductCategory：多对多关系。**

**In this design I use a junction table called ProductCategory to handle the many-to-many relationship between products and categories. The reason is that a single product can belong to multiple categories — for example, an iPhone could be under both Electronics and Mobile Phones. At the same time, each category can also contain many different products. If I tried to put a single foreign key in either the Product table or the Category table, it would only support a one-to-many relationship, which isn’t flexible enough. With a junction table, I can simply insert a row for each mapping, and then it’s easy to query products by category or categories by product.**

**在这个设计里，我用了一个中间表 ProductCategory 来处理商品和分类的多对多关系。原因是，一个商品可能同时属于多个分类，比如 iPhone 既可以属于 Electronics，也可以属于 Mobile Phones；而一个分类也可以包含很多商品。如果我在 Product 表或 Category 表里单独加一个外键，只能表达一对多的关系，无法灵活地覆盖多对多的情况。有了中间表之后，只要在表里插一条对应关系，就能很方便地查某个分类下的商品，或者某个商品所属的所有分类。**

**API 设计（保持一个核心接口）**

**用一个 Get Category API就能把所有层级的读取统一起来；通过 path parameter 和 query parameters 来切换“取根”“取节点”“取直属子类”“取商品列表**

**1. 获取分类数据（统一接口）**

**GET /api/categories/:id?depth=1&include=children,products&limit=20&offset=0**

* **:id = root 时表示顶层分类**
* **depth = 返回多少层（默认 1，不超过 3）**
* **include = children、products，控制是否返回子类/商品**
* **limit / offset = 商品分页**
* **sort = 可选排序，比如 price\_asc、popularity\_desc**

**Response**

**{**

**"category": { "id": 1, "name": "Food", "parent\_id": null },**

**"subcategories": [**

**{ "id": 3, "name": "Fruits", "parent\_id": 1 },**

**{ "id": 4, "name": "Vegetables", "parent\_id": 1 }**

**],**

**"products": [**

**{ "id": 101, "name": "Apple", "price": 1.5, "image\_url": "..." },**

**{ "id": 102, "name": "Banana", "price": 0.8, "image\_url": "..." }**

**],**

**"tree": { "id": 1, "children": [ { "id": 3, "name": "Fruits", "children": [] } ] },**

**"paging": { "limit": 20, "offset": 0, "total": 187, "next\_offset": 20 }**

**}**

**API 版控与格式｜Versioning & Format**

* **Versioning**：路径前缀 /v1；向后兼容时新增字段置于对象末尾。
* **分页**：page + limit 或 cursor（推荐 cursor：更稳）。
* **排序/筛选**：白名单校验，避免 SQL 注入；面向搜索引擎时在 BFF 翻译为 DSL。
* **国际化**：Accept-Language 或 query locale；BFF 负责价格/货币组合。
* **安全**：公开接口只读；管理接口需 OAuth2/OIDC + RBAC；所有写接口支持 Idempotency-Key。

**前端取数逻辑**

* **首页：GET /api/categories/root?depth=1 → 顶层分类**
* **点击某类：GET /api/categories/:id?depth=2 → 该类 + 直属子类 + 子类预览**
* **商品翻页：同接口 + limit/offset**

**为什么这样设计？**

* **简单：只需要一个接口，前端 fetch 模式清晰。**
* **完整：支持顶层、子类、商品列表，功能覆盖需求。**
* **可扩展：支持无限层级，也能限制 depth，避免过大 payload。**
* **易缓存：请求结果可以直接以 (id, depth, include, sort, limit, offset) 为 key 缓存。**

**6️⃣ 可扩展性 & 高可用设计（面试 follow-up）**

**1. Cache**

* **Use Redis to cache /api/categories/:id responses.**
* **Key format: category:{id}:data → value stored as JSON.**

**2. Read-heavy Optimization**

* **Category and Product data is mostly read-heavy and updated infrequently.**
* **Apply read/write separation with database replication.**

**3. Scaling**

* **Load balancer distributes traffic across multiple Web/API servers.**
* **Database sharding or partitioning based on category\_id or product\_id.**
* **Use CDN & S3 to store and deliver product images efficiently：**We store images in S3, but we don’t serve the raw S3 URL to end users. Instead, S3 acts as the origin and a CDN like CloudFront or Cloudflare sits in front to deliver the files. The CDN gives us edge caching, lower latency, better throughput, and cheaper egress at scale. It also lets us add image optimizations and transformations at the edge

**4. High Availability**

* **Database: primary-replica setup with automatic failover.**
* **Redis cluster with persistence enabled.**
* **Web servers with autoscaling to handle traffic spikes.**

**面试中可能被问到的细节**

**用户第一次打开网页 展示顶层 Category + 热门商品 热门商品可以提前缓存 fetch下两层 同时存入cache 这样 读取速度快 不需要reload api 也不需要一次把数据全拿出来对浏览器造成损害**

For the landing page I’d keep it lightweight: just fetch the top-level categories and maybe a few featured products. Then, when the user clicks into a category, I’d allow the API to return not only the direct children but also the next one or two levels in a summary form. That way, when the user drills further, the data is already cached on the client side and we don’t need another round trip. This approach avoids overloading the browser with the full tree, but still reduces the number of API calls and improves perceived performance.

首页我会保持轻量，只取顶层分类和少量热门商品。用户点击某个分类时，API 不仅返回这个分类的直属子类，还可以预取下面 1–2 层的子类，用精简的数据形式返回。这样用户再往下点的时候，前端直接从缓存里拿，不用再次请求。这样避免了一次性把整棵树都打包过来导致浏览器卡顿，同时又减少了 API 请求次数，提高了响应速度。

**商品更新 / 分类更新 可以使用消息队列（Kafka / RabbitMQ）通知 cache 更新**

When a product or category gets updated, I don’t want every web server to directly hit the database and refresh caches individually, because that can create heavy load and even inconsistencies. Instead, I’d publish an event to a message queue like Kafka or RabbitMQ. The consumers that subscribe to this event—say, a cache service—will receive the update and either invalidate or refresh the relevant Redis keys. This way the cache stays in sync with the database, the update is propagated quickly to all nodes, and the system remains decoupled and scalable.

当商品或分类更新的时候，我不希望每个 Web 服务器都直接去查数据库然后各自刷新缓存，这样会增加负担，也可能导致数据不一致。更好的方式是把更新事件发布到消息队列，比如 Kafka 或 RabbitMQ。订阅这个事件的消费端（例如缓存服务）就会收到通知，然后去失效或者刷新对应的 Redis key。这样缓存就能和数据库保持同步，更新能快速传播到所有节点，同时系统也更加解耦、可扩展。

**管理员视角：管理 Category / Product / ProductCategory & API: POST/PUT/DELETE 增删改**

管理员视角设计（Admin View）

功能

* 管理 Category（分类）
* 管理 Product（商品）
* 管理 ProductCategory（商品-分类映射关系）

API（REST 风格）

* Category
  + POST /api/admin/categories → 新增分类
  + PUT /api/admin/categories/:id → 更新分类
  + DELETE /api/admin/categories/:id → 删除分类
* Product
  + POST /api/admin/products → 新增商品
  + PUT /api/admin/products/:id → 更新商品
  + DELETE /api/admin/products/:id → 删除商品
* ProductCategory
  + POST /api/admin/product-categories → 新增映射（把某商品加到某分类）
  + DELETE /api/admin/product-categories → 删除映射（把某商品从某分类移除）

数据流

* 管理员写操作 → 主库写入
* 成功写入后 → 通知缓存失效（Redis）
* 通知索引服务（如 Elasticsearch）异步更新

From the admin perspective, I’d expose a very simple set of CRUD APIs for managing categories, products, and their mappings. For example, admins can create, update, or delete categories through endpoints like POST /api/admin/categories or PUT /api/admin/categories/:id. The same applies for products. For the many-to-many mapping, I’d provide a POST and DELETE on /api/admin/product-categories so that an admin can assign or remove a product from a category. Each write goes to the primary database, and then we publish an event to invalidate the relevant Redis cache and update the search index asynchronously. This keeps the browsing side consistent while giving admins a straightforward way to maintain the catalog.”

从管理员的角度看，我会提供一组非常简单的 CRUD API，用来管理分类、商品和它们之间的映射关系。比如管理员可以通过 POST /api/admin/categories 或 PUT /api/admin/categories/:id 来创建或更新分类，商品的接口也类似。对于多对多关系，我会提供 POST 和 DELETE 的 /api/admin/product-categories 接口，让管理员可以把商品加入分类或从分类中移除。每次写操作都会先写到主库，然后触发事件通知去失效 Redis 缓存，同时异步更新搜索索引。这样既保证了前端浏览的一致性，又给管理员提供了一个简单直观的方式去维护商品目录

**💡 总结逻辑：**

1. **Category 用 self-join 实现多级树**
2. **Product + ProductCategory 实现多对多**
3. **单个API 清晰暴露顶层 Category、子类 + 商品**
4. **前端 fetch pattern 简单直接，避免深递归**
5. **可扩展性靠 cache + load balancer + DB replication**

**Q1. 如何查询顶层分类？**

**EN:**  
SELECT … FROM category WHERE parent\_id IS NULL AND is\_active = TRUE ORDER BY display\_order, id LIMIT …;

**ZH:**  
SELECT … FROM category WHERE parent\_id IS NULL AND is\_active = TRUE ORDER BY display\_order, id LIMIT …;

**Q2. 如何查询某分类的直属子类和商品？**

**EN:**

* **Children:** SELECT … FROM category WHERE parent\_id = :id AND is\_active = TRUE ORDER BY display\_order, id;
* **Products (direct):** join product\_category + product, filter active, order by popularity/price, and paginate.

**ZH:**

* **子类：** SELECT … FROM category WHERE parent\_id = :id AND is\_active = TRUE ORDER BY display\_order, id;
* **商品（直属）：** product\_category 连接 product，只取 is\_active，按 popularity/price 排序并分页。

**Q3. 如何扩展到大访问量？**

**EN:**

* **Horizontal scale** stateless BFF/Query/Admin behind a load balancer.
* **DB**: primary + multiple **read replicas** (geo-distributed); read traffic routes to nearest replica.
* **Search**: multi-node cluster (sharding + replicas).
* **Cache**: Redis cluster with partitioning and persistence (AOF/RDB).
* **CDN**: global POPs for static and selected JSON.

**ZH:**

* **横向扩展**无状态 BFF/Query/Admin，前置负载均衡。
* **数据库**：主库 + 多只读副本（可多地域），读请求就近路由。
* **搜索**：多节点集群（分片 + 副本）。
* **缓存**：Redis 分片集群并开启持久化。
* **CDN**：全球节点缓存静态与部分 JSON。

**Q4. 如何存储分类（自关联树）？**

**English**  
“I model categories with a self-join table. Each row has an id and a parent\_id that points back to the same table; parent\_id = NULL means it’s a root category. That keeps the schema simple and flexible—whether we have three levels or unlimited depth, we don’t need to change the schema. Direct-children queries are trivial with WHERE parent\_id = ?, and we can use a recursive CTE （Common Table Expression） if we ever need a bounded subtree.”

**中文**  
“我用自关联表来存分类。每行有 id 和指向同表的 parent\_id，parent\_id = NULL 表示顶层。这样结构既简单又灵活——无论三层还是无限层都不用改表。查直属子类非常简单，WHERE parent\_id = ? 就行；如果需要拿到受限深度的子树，可以用递归 CTE。”

**Q5. 如何存储商品与分类的多对多关系？**

**English**  
“I use a junction table ProductCategory(product\_id, category\_id) with a composite primary key. That lets one product appear in multiple categories and each category contain many products. It also gives me clean queries in both directions—‘products in a category’ and ‘categories for a product’—and I index (category\_id, product\_id) for the hot path.”

**中文**  
“我用中间表 ProductCategory(product\_id, category\_id)，复合主键处理多对多。这样一个商品可以属于多个分类，一个分类也能包含很多商品；查询双向都很干净——按类取商品、按商品查分类。同时对 (category\_id, product\_id) 建索引，覆盖最常见的读取路径。”

**Q6. 商品属于多个分类怎么处理？**

**English**  
“That’s exactly why we have the junction table. Each mapping is just one row in ProductCategory. When I list a category, I join by that category\_id, so a product can legitimately show up under multiple categories without duplication problems in storage. Only if the UI mixes multiple categories in one view would I dedupe on the client.”

**中文**  
“这正是中间表存在的原因。ProductCategory 里每条映射就是一行。按 category\_id 查询某个分类时，商品自然可以在多个分类下被展示，存储层没有重复问题。只有当前端在一个视图里混合多个分类时，才需要在前端做去重。”

**Q7. 如何缓存分类与商品？**

**English**  
“I cache category nodes and product lists in Redis using predictable keys like cat:{id}:d{depth}:inc{include}:sort{…}:limit{…}:offset{…}. Popular nodes also get edge-cached via CDN. TTLs are short—say 5–15 minutes for product pages—and we do precise invalidation on updates so users see fresh data without hammering the database.”

**中文**  
“我把分类节点和商品列表缓存到 Redis，键可预期，比如 cat:{id}:d{depth}:inc{include}:sort{…}:limit{…}:offset{…}；热门节点再配合 CDN 做边缘缓存。TTL 设短一些（商品页 5–15 分钟），同时在数据更新时做**精准失效**，既保证数据新鲜，又避免数据库被打爆。”

**Q8. 如何保证系统稳定运行？**

**English**  
“I keep the services stateless behind a load balancer, push reads to database replicas, and lean on Redis and CDN for hot paths. We have rate limiting and circuit breakers at the gateway, plus clear fallbacks—serve cached data if search degrades, or reduce page size if cache is down. Observability tracks cache hit ratio and P95 latency so we can catch regressions early.”

**中文**  
“我把服务做成无状态挂在负载均衡后面，读流量尽量走数据库只读副本，热点路径依赖 Redis 和 CDN。网关层做限流与熔断，并准备好降级方案——搜索异常时先用缓存，缓存异常时减小分页。可观测性重点看缓存命中率和 P95 延迟，及时发现回归和瓶颈。”

**Summary：**

So the goal of this system is to let users browse products by category. When they first open the page, they’ll see all the top-level categories like Food or Clothes, and when they click into one, the system should return both the direct subcategories and the products under that category. Because a product can belong to multiple categories, the schema is centered on three tables: Category, which is a self-join table so each category just points to its parent; Product, which stores the items themselves; and ProductCategory, which manages the many-to-many relationship. This makes the queries very straightforward: one query for top-level categories, another for a category’s direct children, and another join to get the products inside that category.

On the API side, I keep it even simpler by exposing a single Get Category API that can handle all these cases. By adjusting the path parameter and query parameters, the same endpoint can return root categories, a specific category node, its direct subcategories, or a paginated list of products. That makes the front-end fetching pattern very predictable and easy to work with—every click just maps to one call to the same endpoint, without recursion or complicated traversal.

Since the system is read-heavy, caching is critical. Category responses and product lists can be cached in Redis, and a CDN can be used for static assets and JSON for popular categories. For database scaling, read replicas absorb most of the query traffic. Combined with load balancers, caching layers, and database replicas, this architecture can handle high user traffic while still delivering fast responses and consistent data to users.

**Requirement Clarification**：

Before diving into the design, I’d like to clarify scope so I target the right solution. First, is this a read-only browsing experience, or do we also need admin CRUD for categories and products? For categories, are they unbounded depth—like infinite levels—or do we have a fixed maximum depth? When users drill down, should we only return direct children per click, or would you prefer prefetching the next 2–3 levels for faster navigation? On the caching side, do you expect us to cache these results at the edge via CDN, in Redis, or both?

In terms of the data we need to display, I assume it includes:

* Top-level categories when the page loads,
* Direct subcategories when a category is expanded,
* The list of products under that category, with pagination and optional sorting,
* And possibly featured or popular products on the landing page.”

在开始设计之前，我想先澄清下范围，确保方案对题。首先，这个系统是只读浏览，还是也要支持管理员对分类和商品的 增删改？分类层级是无限还是有固定的最大层数？用户下钻时，是每次只返回直属子类，还是希望预取下 2–3 层来加快体验？在缓存上，结果是要放在 CDN 边缘还是 Redis，或者两者都用？

至于需要展示的数据，我理解主要包括：

* 页面加载时展示的顶层分类，
* 点击分类时展示的直属子类，
* 该分类下的商品列表（支持分页和可选排序），
* 以及首页可能需要的精选或热门商品。

High Level + API + DB Schema + Cache / Read Replica + Scalability + Security + Deployment