
復旦大學

# 商业决策平台的智能数据管理系统

**卢羿舟**

统计与数据科学系

2025 年 5 月 21 日

# 目录

# 项目背景

- 在当今快速发展和竞争激烈的商业环境中，企业对于精准、高效的决策能力的需求日益增长。为了应对这一挑战，构建一个强大的商业决策平台显得尤为重要
- 你是一家新兴的“数据驱动商业决策平台”的员工，公司要求你帮助设计其底层数据处理模块。你需要设计一个系统支撑平台的智能化运行

# 问题回顾

该系统存储的数据包括

- 营销任务与任务之间的依赖关系
- 表示客户之间关系的加权有向图
- 商品数据

需要完成的功能包括

- 营销任务优先调度功能
- 客户网络与影响力传播分析
- 商品数据检索

# 目录

# 整体规划

根据要求，将系统拆分成三个模块

- 营销任务优先调度模块
- 客户网络与影响力传播分析模块
- 商品数据检索模块

# 本节目录

# 模块功能拆解

目标：实现一个存储所有营销任务的数据结构，支持

- 对营销数据的增、删、改、查
- 根据任务之间的相互依赖优先执行优先级最高的任务
- 展示任务优先级最高的 $k$ 个任务

# 核心数据结构与规划逻辑

- 模块要求我们实现可以高效查找优先级最高的营销任务，同时还需要展示优先级最高的 $k$ 个任务，优先队列可以很好的完成这个问题，因此该模块首选**优先队列**作为核心数据结构
- 在优先队列中，考虑且仅考虑当前不存在前置依赖的任务，存在前置依赖的任务会被移除出优先队列，而新增的不存在前置依赖的任务会被加入优先队列

# 任务依赖表示

为了考虑任务之间的前置依赖关系，我们需要实现一个有向图，规定

$$\text{前置任务} \longrightarrow \text{后置任务}$$

为有向图的基本结构，同时为了避免出现相互依赖的问题，需要额外要求该有向图为有向无环图

# 任务状态管理

为了更好的管理任务状态，考虑为任务附加一个属性 `_status`，用来表示任务当前的状态。我们将任务的状态分成三类：

- `Pending`
- `Ready`
- `Complete`

分别代表三种状态：该任务存在前置依赖，该任务不存在前置依赖与该任务已完成。

# 任务 ID 设计

考虑可能会修改任务名，为了实现对任务的唯一识别，额外添加 `_task_id` 属性，它满足

- 唯一性
- 不变性

在每一个营销任务对象被创建时就赋予其任务 ID，设计格式为“时间戳-UUID[1]”，这样设计的好处是

- 在并发条件下 ID 不冲突
- ID 存在一定的有序信息，方便检索和管理

[1]一个唯一的 32 位十六进制字符串

# Top-k 功能设计

由于没有提前指定 $k$ 的具体数量，所以无法直接在系统运行中提前规划大小为 $k$ 的优先队列，为了实现这一功能，有两个设计思路

1. 每次调用时，优先队列连续弹出 $k$ 个节点，记录信息后再插入优先队列恢复
2. 复制一份优先队列，随后连续弹出 $k$ 个节点后删除

为了避免可能过高的空间复杂度，最终采用了方案 1

# 优先队列及任务状态管理逻辑

一个营销任务在其生命周期中，可能会经历以下状态改变：

- 初始化状态为 `Ready`（由于不存在前置依赖）
- 由于被添加了前置依赖，状态由 `Ready` 变为 `Pending`
- 由于前置依赖被完成或删除，状态由 `Pending` 变为 `Ready`
- 处于优先队列队首被执行，状态由 `Ready` 变为 `Complete`

对于每一次的状态改变，都会影响其本身和其后续任务在优先队列中的状态，此外，倘若修改处于 `Ready` 状态的任务优先级，也会影响其在优先队列中的位置，这需要我们能够高效调整优先队列，但。。。

# 支持高效查找的优先队列设计

为了解决优先队列无法实现高效查找与删除的问题，该模块在以数组为基础的优先队列的基础上，额外维护一个字典，字典中的键值对为

$$\{\texttt{task\_id} : \text{堆中索引}\}$$

用于高效查找目标任务在堆中的位置，实现高效的更新与删除

# 功能实现——MarketingTask 类

MarketingTask 类有以下属性：

- _task_id
- _name
- _status
- urgency
- influence
- _priority

使用了 __slots__ 命令来优化存储空间（因为这个类会被大量实例化），同时利用 @property 和 setter 来控制对这些私有变量的访问和修改，并在每次修改后自动更新优先级

# 功能实现——MarketingTask 类

此外，MarketingTask 类还包含两个私有方法，用于更新任务的属性和状态

- _update_details()[2]：更新任务的紧急程度或者影响力，并更新优先级，如果某个参数为 None，则不更新该属性，如果更新成功，则返回 True，如果没有更新，则返回 False
- _set_status()：设置任务的状态

[2]注意，由于美观性以及空间因素，在 slides 中所有出现的函数都默认不填写参数名和参数类型，详细的参数说明可以参考源代码注释。

# 功能实现——TaskDependencyGraph 类

为了实现高效的查找节点相互之间的依赖关系，该类总共维护了两个字典，分别储正向依赖关系以及反向依赖关系，以及一个集合用于存储任务 ID。其中正向依赖关系指的是从某个任务节点指向其他后续任务的映射，而反向依赖关系指的是某个任务节点被哪些任务前置影响的映射。它包含了以下图的基本方法

- `add_task()`：向图中添加一个任务节点
- `remove_task()`：从图中移除一个任务节点及其所有相关依赖，并返回这个节点 id
- `add_dependency()`：添加依赖关系：prerequisite_id -> dependent_id，在添加前会进行环路检测
- `remove_dependency()`：移除依赖关系：prerequisite_id -> dependent_id
- `get_dependents()`：获取直接依赖于 task_id 的任务集合
- `get_prerequisites()`：获取 task_id 的前置任务集合
- `has_task()`：检查任务是否存在于图中
- `get_all_tasks()`：返回图中所有任务的列表

# 功能实现——TaskDependencyGraph 类

注意，由于我们要求图本身无环，所以在每次添加边的时候，需要对图进行环路检测。这个检测的原理是通过深度优先搜索寻找是否存在一条待添加边的反向通路，倘若存在，则不允许添加这条边，否则会导致图中出现环。这个深度优先搜索通过一个私有方法 `_has_path()` 实现。这个方法通过列表模拟栈来实现了一个迭代式的深度优先搜索

# 功能实现——UpdatableMaxHeap 类

与常规利用列表实现的最小堆不同的是，这个类额外维护了一个字典用于支持节点在列表中的索引查找，从而实现高效的查找和删除。由于堆中元素位置的变动完全由 _swap() 实现，所以只需要在该私有方法中加入更新索引字典的逻辑即可。下面展示公共接口列表

- insert()：向堆中插入一个新任务，如果任务已存在，则更新优先级
- peek_max()：查看堆顶元素，不移除
- extract_max()：移除并返回堆顶元素
- is_empty()：检查堆是否为空
- update_priority()：更新堆中已存在任务的优先级
- delete()：从堆中删除指定任务 ID 的元素
- get_heap_size()：返回堆中元素的数量

# 功能实现——TaskManager 类

这个类是营销任务优先调度模块的核心类，用于实现所有营销任务优先调度模块需要实现的功能。它维护了多个数据结构：

- `_tasks`：字典，任务 ID 到任务对象的映射
- `_task_graph`：`TaskDependencyGraph` 类，储存任务之间的依赖关系
- `_ready_queue`：`UpdatableMaxHeap` 类，储存所有状态为 Ready 的任务的优先队列
- `_in_degree`：字典，状态为 Pending 的任务 ID 到其前置依赖数量的映射

# 功能实现——TaskManager 类

这个类的公共接口

- `add_task()`：往任务管理器中添加一个任务
- `add_dependency()`：添加一个任务间的依赖关系，如果添加的依赖会导致后继任务不再 ready，则需要将其从 ready 队列中移除
- `remove_dependency()`：移除一个已经存在的依赖关系，如果移除成功且导致后继任务的状态变为 ready，则更新 ready 队列
- `mark_task_as_completed()`：将指定任务标记为已完成，同时更新其后续任务的入度字典
- `execute_next_highest_priority_task()`：从 ready 队列中提取优先级最高的任务，并执行
- `get_top_k_ready_tasks()`：查看当前 ready 队列中的优先级最高的 k 个任务
- `update_task_info()`：更新现有的任务信息
- `delete_task()`：从系统中彻底删除一个任务及其所有相关依赖

# 本节目录

# 模块功能拆解

这个模块需要实现对客户关系网络的分析。首先我们定义所有客户与相互的影响关系被储存在一个有向图（这里的图可以有环）中，每个节点均表示一个客户。他们之间的影响关系如下

$$\text{客户}A \xrightarrow{0.8} \text{客户}B$$

代表客户 A 可以直接影响客户 B，而边权代表客户 A 对客户 B 的影响程度，这个数值的取值范围为了分析方便被限制在了 $(0, 1]$，数值越大代表影响程度越深，从而依靠这种关系来构建整个有向图。在这个模块中总共有两个分析任务：

1. 客户重要性评价
2. 寻找每一位客户所能影响到的所有客户

# 客户重要性评价

为了识别客户关系网中最重要的节点，客户网络与影响力传播分析模块实现了两个判断节点重要性的算法

1. 度中心性
2. PageRank

# 度中心性

要衡量一个图中节点的重要性，一个很直观的方法就是看他的入边和出边有多少。这类似于评价一个交通枢纽的重要程度，它能够连接起越多的路，一般而言就越重要。这个评价指标就被称为加权度中心性（**W**eighted **D**egree **C**entrality），加权度中心性越高，代表节点的重要性越高。在有向赋权图中，这个指标进一步变为加权入度中心性（WIDC）和加权出度中心性（WODC），这是为了区分不同边的性质和不同的边权，具体计算方式如下

$$WIDC(o) = \sum_{u \in in(o)} w(u, o), \quad WODC(o) = \sum_{u \in out(o)} w(o, u),$$

其中 $in(o)$ 和 $out(o)$ 分别代表节点 $o$ 的所有入射节点和出射节点。

# 度中心性的问题

度中心性存在以下问题

- 仅考虑相邻节点之间的影响
- 所有节点对影响力的贡献是等价的

然而，在我们的例子中，客户 A 如果能通过客户 B 而对 C 有影响，那么客户 A 的重要程度也应该能传导到对客户 C 的重要程度的评价上。如果一个客户能同时影响更多的客户，它的影响力应该和只能影响很少客户的影响力有所不同

# PageRank

PageRank (Page u. a., 1999) 最开始被用于评价网页重要性，其核心思想基于这样一种直观的理念：一个网页的重要性取决于指向它的其他网页的数量和质量。简单来说，PageRank 算法模拟了一个在互联网上随机冲浪的用户。用户从一个随机的网页开始，然后不断地点击页面上的链接。一个网页被访问的概率越高，那么这个网页就被认为越重要。

# 影响力范围分析

为了综合考虑两个节点之间的距离和边权重对影响程度的影响，引入路径影响力概念，具体而言，对于节点 $u$ 和节点 $v$，设这两个节点之间的路径集合为 $\mathcal{L}$，则这两点之间的路径影响力定义为

$$IF(u,v) = \max_{l \in \mathcal{L}} \prod_{o,o' \in l} w(o,o'),$$

即为路径途径所有边的边权乘积中的最大值。由于之前限制了边权的取值范围为 $(0,1]$，所以最终的路径影响力取值范围也应该为 $(0,1]$，且影响力值随着路径延长单调递减，我们可以很容易设置一个截断阈值来获得在一定范围和影响力内的客户影响力集合。

# 功能实现——CustomerGraph 类

该类为一个允许环路的有向赋权图，使用邻接列表来存储，该数据结构存在以下公共接口

- `add_customer()`：向图中添加一个新客户（节点）如果客户已存在，则不执行任何操作
- `get_all_customers()`：返回图中所有客户名称的列表
- `add_influence()`：在两个客户之间添加一条有向的影响力关系（边）如果'from_customer' 或'to_customer' 不在图中，会先将它们添加进来
- `get_direct_influencees()`：获取一个客户直接影响的所有其他客户及其对应的影响力权重
- `get_influence_weight()`：获取从 from_customer 到 to_customer 的直接影响力权重

# 功能实现——算法

基于 CustomerGraph 类，实现了以下四个算法：

- calculate_weighted_out_degree_centrality()
- calculate_weighted_in_degree_centrality()
- calculate_pagerank()
- analyze_all_customer_influence_nodes()

# 本节目录

# 模块功能拆解

在这个模块中，我们需要维护一个管理所有商品的结构，需要实现的功能如下：

- 高效插入、删除、更改商品信息和搜索
- 搜索任意价格范围内的所有商品
- 根据商品名称前缀进行前缀搜索

同时需要考虑过多的商品无法一次性加载进入内存查找，需要对外存查找进行优化

# 平衡二叉搜索树的外存低效

想象在一个很厚的有序字典（数据）中找一个词，字典非常厚以至于分成了上万本存放在图书馆（硬盘）里，你的书桌（内存）无法放下所有的字典，一次只能取一本查找，所以可能要频繁去书架上找下一次搜索到的字典（多次磁盘 I/O 操作），每次只能拿回来一本书，而去书架取书太慢了，所以会造成查找低效

# B+ 树

为了提高外存搜索效率，有两个可以优化的方法

- 降低去书架找书的次数
- 相近的词一定连续地存放在同一个字典中
  (外存中的数据存储一般是离散的)

B+ 树优化了这两个方向，进而实现了高效的外存查找

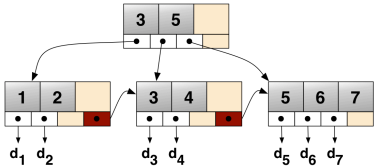


图: B+ 树的基本结构

# 前缀搜索

Trie 是一种天然支持前缀搜索的树，其每一个节点都代表字符串中的某一位字符，因此如果想要实现前缀搜索，只需要搜索到前缀末尾的那个字符对应的节点，然后遍历即可。

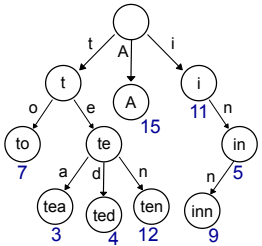


图: 一个保存了 8 个键的 trie 结构，"A"，"to"，"tea"，"ted"，"ten"，"i"，"in"，"inn"

# 商品 ID 设计

与此前 `MarketingTask` 相同，我们需要一个对商品对象的唯一标识，设计格式为“PROD-时间戳-UUID”，目的与此前一样，且可以与营销任务的 ID 做明显的区分

# 数据存储逻辑设计

回顾我们总结的模块功能，我们需要至少实现通过价格和商品名进行查找，然而商品价格和商品名都有可能发生更改和重复。这在数据存储中是极为低效的，意味着我们需要存储两套商品对象的副本，出现大量的存储空间浪费，为此，在该模块借用数据库设计理念，作出如下设计

- **主键**：为商品对象的唯一 ID，可以通过主键查找商品对象
- **次键**：包括商品名和商品价格等，可以通过次键查询对应的商品 ID

这样实现了

1. 优化存储空间
2. 提高数据一致性和维护效率

# 功能实现——Product 类

与 MarketingTask 类类似，Product 类通过 @property 和 setter 来规定对其属性的访问和修改操作，同时对于这种会大量实例化的类，通过 __slots__ 进行存储优化，其包含如下属性

- _product_id
- name
- price
- heat

# 功能实现——B+ 树

由于模块中需要有两种不同的 B+ 树，一种键和值是一一对应的关系（ID-Product 对象），一种一个键可以对应一系列值（价格-ID）。两种 B+ 树在搜索、插入、删除等逻辑上存在细微差别，因此，首先定义了 `BaseBPlusTree` 作为基类，规定了 B+ 树的一系列底层操作，模块中使用的两种 B+ 树均继承自这个基类。关于 B+ 树，总共实现了以下四个类

- `BPlusTreeNode`：B+ 树节点类，定义内部节点和叶节点的行为
- `BaseBPlusTree`：B+ 树基类，定义 B+ 树的底层操作
- `BPlusTreeProducts`：用于存储（价格-ID）对，支持范围查找
- `BPlusTreeID`：用于存储（ID-Product 对象）对

# 功能实现——Trie 树

第二个次键是商品名，用来通过商品名前缀或者全程来搜索对应商品的逻辑。这部分实现了以下两个类

- `TrieNode`：节点类，标记该节点是否为单词末尾，若是，存放对应商品 ID
- `ProductPrefixTrie`：实现了一个 Trie 树，支持插入、删除和前缀匹配

与标准 Trie 树不同的是，普通 Trie 树在删除时仅仅删除节点上的标记，这会导致大量的存储空间浪费，为此，在实现 Trie 树的 delete 方法时，额外实现了一个基于回溯的节点删除逻辑，回收不再用于索引商品名的冗余节点

# 功能实现——ProductManager 类

这个类用于实现该模块的所有功能，包括

- 添加、删除、更新商品信息
- 通过 ID 精确搜索商品
- 通过价格进行模糊搜索和范围搜索
- 通过前缀进行模糊搜索（仅返回 k 个热度最高的商品）
- 通过全称进行搜索

其使用 `BPlusTreeID` 存储 ID 作为主键，使用 `BPlusTreeProducts` 存储价格作为次键，使用 `ProductPrefixTrie` 存储名称作为次键，实现高效的索引

# 目录

# 关于营销任务优先调度模块

在考虑营销任务前置依赖的优先调度时，该模块目前仅仅考虑在不存在前置依赖的任务中进行优先度排序，从中选择优先度最高的执行。然而，在实际环境中，可能存在某个前置任务虽然不紧急，但是其后置任务十分紧急的情况，而当前的算法无法捕捉这种后继任务的重要性。

**解决方案**：由于规定了营销任务之间的依赖图不能有环，而任何一个有向无环图可以通过若干条不相交的路径进行路径覆盖，因此考虑将依赖图拆分成多条路径，对于每一条路径进行整体的优先级排序。

# 路径优先级

定义路径优先级，首先定义一个衰减因子[3]$\gamma$，则路径中某个节点 $o$ 对路径优先级的影响力可以计算为

$$IF(o) = \gamma^k \text{priority}(o),$$

其中 $k$ 为路径起点到节点 $o$ 的步数，衰减因子可以由任务的时效性决定，时效性越强的任务，$\gamma$ 越接近于 $1$。通过对路径优先级进行排序，来决定优先执行哪一条路径的第一个任务。当有任务被完成时，重新计算路径优先级。

[3]本质是一个惩罚项，越靠后的高优先级任务需要提前完成的代价越大。

# 关于客户网络与影响力传播分析模块

这部分功能主要有以下三个改进方向

1. 当前仅仅采用了度中心性和 PageRank 来评价节点重要性，但是并未有一个综合的统计分析，未来可以继续添加其他的评价指标，然后构建统计模型进行综合分析
2. 关于最小影响力阈值，对于不同结构不同节点数量的图，理想的阈值应该不一样，现在主要依靠经验制定，后续可以使用一些指标进行自适应计算
3. 当前处理图数据的主要思路是 Offline 的，但是在真实业务场景中，数据往往是 Online 的，可以通过缓存已有的路径信息来提升运行效率，降低计算开销

# 关于商品数据检索模块

当前使用 Trie 树进行前缀匹配，然而 Trie 树在外存的效率并不高，后续可以考虑将 B+ 树和 Trie 树结合起来，使用 B+ 树的每一个叶节点存放一个 Trie 树的方式，来提升前缀匹配效率

# 源代码

https://github.com/FDULeolu/data_structure_pj
感兴趣的同学可以帮忙点点 Star，谢谢！

# References

[Page u. a. 1999] Page, Lawrence ; Brin, Sergey ; Motwani, Rajeev ; Winograd, Terry: The PageRank Citation Ranking : Bringing Order to the Web. In: *The Web Conference*, URL https://api.semanticscholar.org/CorpusID:1508503, 1999