缓冲区和存储管理器实现

实验目的

本实验的目标是实现一个简单的存储管理器（`DSMgr`）和缓冲区管理器（`BMgr`），并通过trace-driven实验来测试和分析其性能。这个实验旨在深入理解数据库系统在存储和缓冲管理方面的基本概念和技术，包括页面读写、缓冲策略和替换算法。

实验环境

- **操作系统**: Linux
- **编程语言**: C++
- **依赖库**: 标准模板库 (STL)
- **工具**: Visual Studio Code, g++, GNU Make

实验内容及步骤

步骤 1: 环境搭建

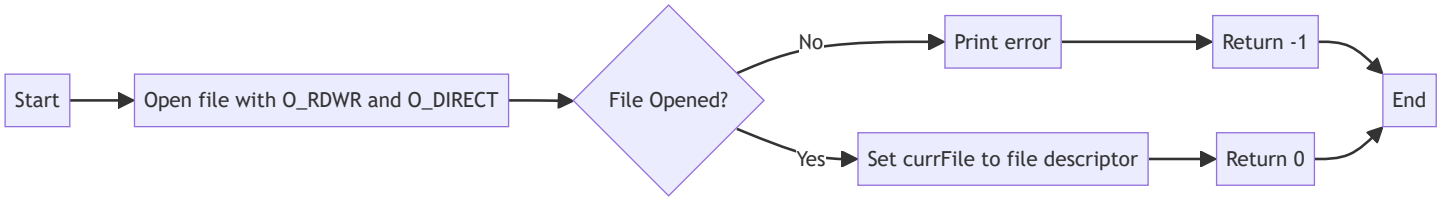
环境搭建包括在Linux系统上安装必要的编译工具，如g++和Make，以及配置Visual Studio Code作为开发环境。

步骤 2: 存储管理器 (DSMgr) 实现

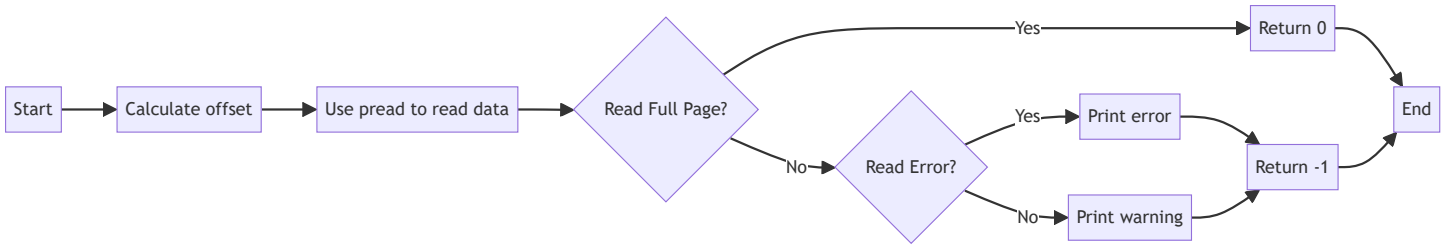
实现了 `DSMgr`` 类来管理磁盘上的页面存储。主要方法包括：

- `OpenFile`：打开数据文件。
- `CloseFile`：关闭数据文件。
- `ReadPage`：从磁盘读取指定的页面。
- `WritePage`：将页面写入磁盘。

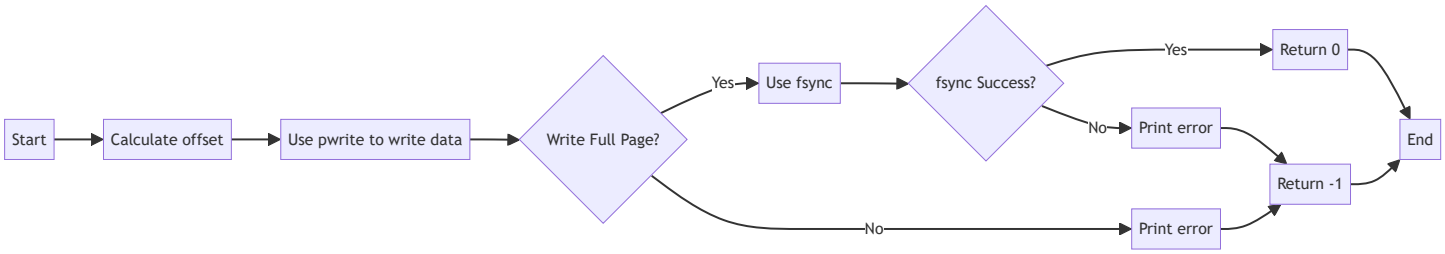
OpenFile



ReadPage



WritePage

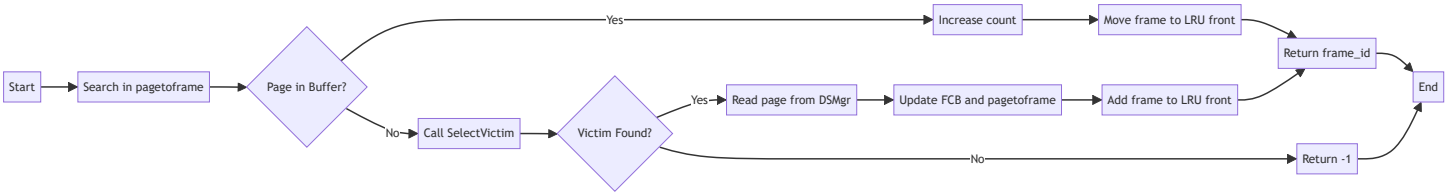


步骤 3: 缓冲区管理器 (BMgr) 实现

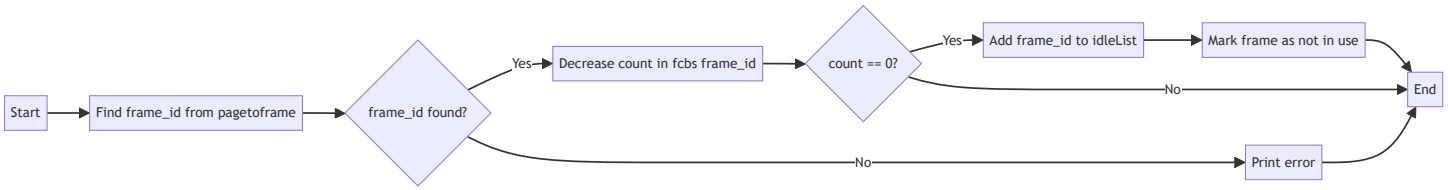
实现了 `BMgr` 类来管理缓冲区。主要方法包括：

- `FixPage`：根据页面ID和操作类型（读/写）固定页面。
- `FixNewPage`：分配一个新页面。
- `UnfixPage`：解固定指定的页面。
- `SelectVictim`：选择牺牲页以供替换。

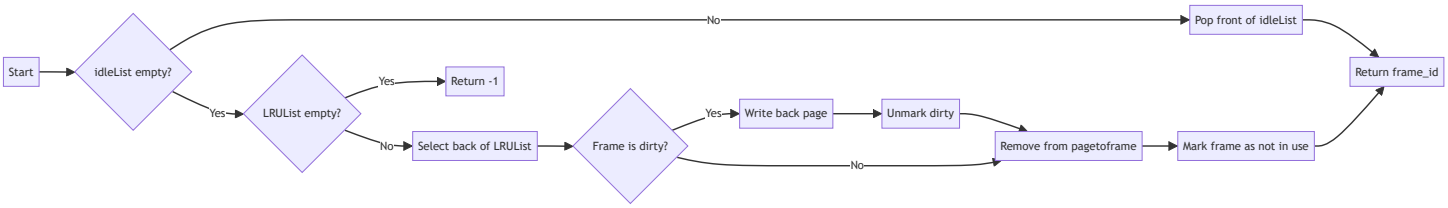
FixPage



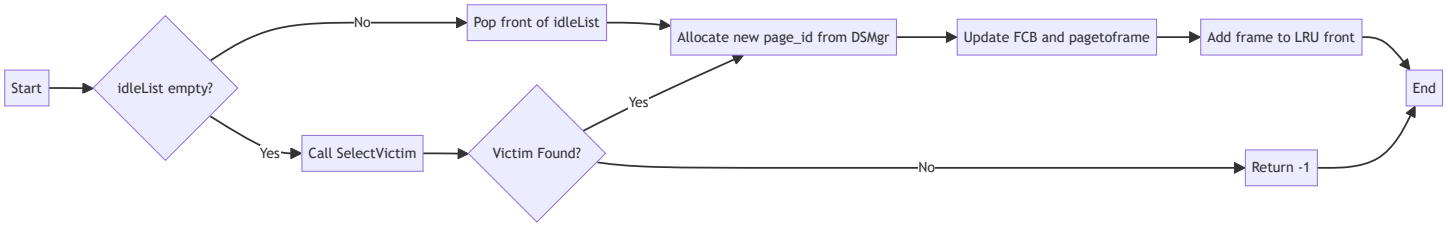
UnfixPage



SelectVictim



FixNewPage



步骤 4: 实验测试

实验测试包括初始化数据库文件和执行trace-driven测试。

1. 初始化数据库文件:
   使用 init 函数在磁盘上创建了50,000个空白页面，并在每个页面的开头写入了 page_id。
   命令行输入并执行如下命令

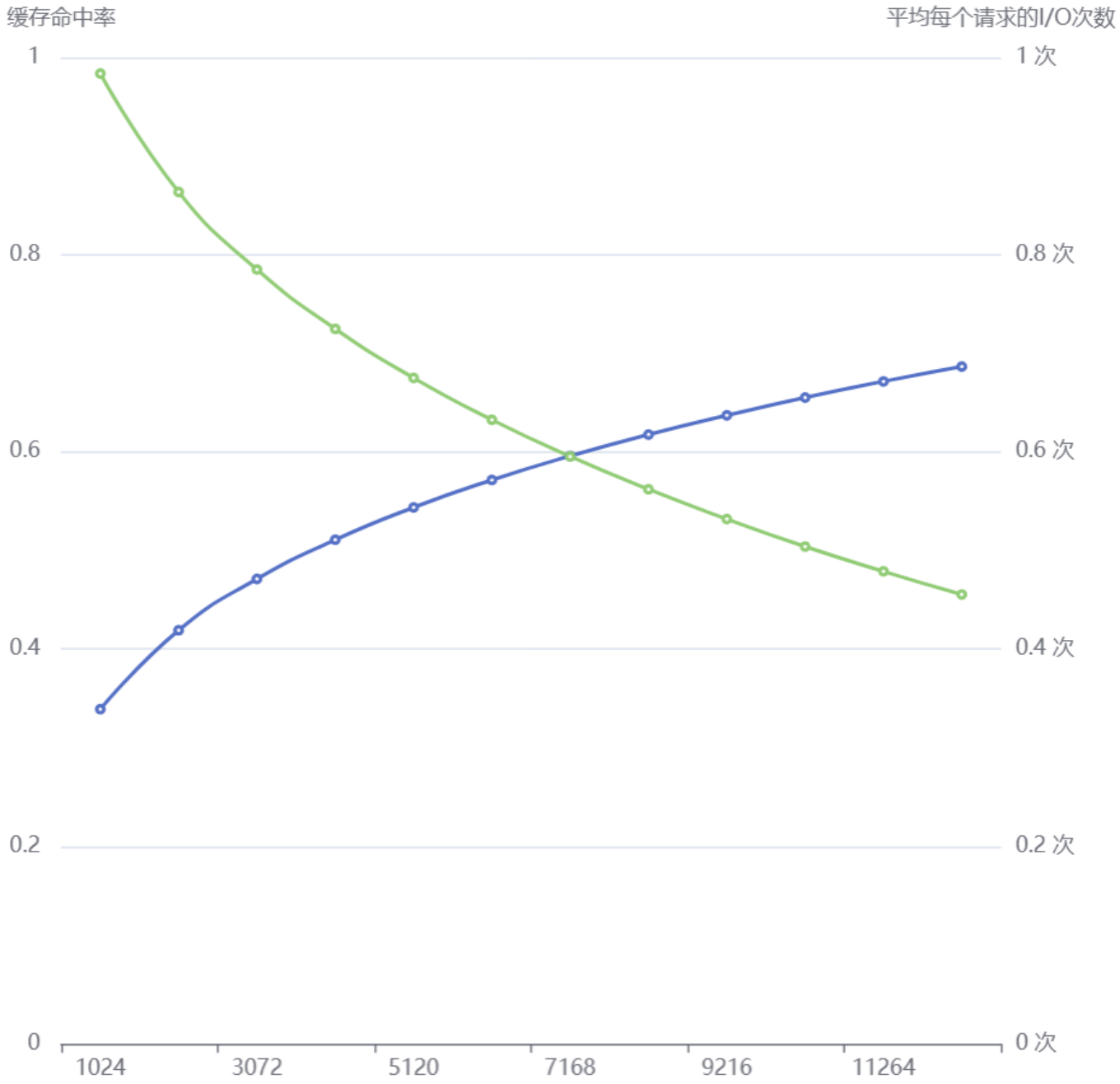
```
make init
```

3. 执行trace-driven测试:
   按照 Zipf 分布生成的页面访问序列，使用 BMgr 类的方法模拟页面的读取和写入操作。记录了内存与磁盘之间的总I/O次数，以评估缓冲区管理器的性能。
   命令行输入并执行如下命令

```
make test
```

实验结果

- 缓冲区大小与命中率和I/O次数



结果分析

随着缓冲区页面总数 BUFFSIZE 的增大，观察到以下变化：

- 磁盘I/O数量的减少：
  - 更多的页面可以被缓存：当 BUFFSIZE 增加时，缓冲区可以容纳更多的页面。这意味着在处理页面请求时，有更高的可能性在缓冲区中找到所需的页面，从而减少了对磁盘的访问。

    - 减少页面替换：更大的缓冲区意味着较少的页面替换。页面替换通常涉及将脏页写回磁盘，并从磁盘读取新的页面到缓冲区中。当缓冲区足够大，能够容纳更多的活跃页面时，页面替换的频率会减少。
- 命中率的增加：
    - 提高数据局部性：缓冲区越大，对数据的局部性（数据访问的时间和空间局部性）的利用就越好。如果一个数据项被访问，那么与其相邻的数据项在不久的将来也有可能被访问（时间局部性），同时在缓冲区中可以连续存储更多的数据项（空间局部性）。
    - 减少冷启动影响：缓冲区较大时，启动阶段填充缓冲区的影响相对于整个缓冲区的使用会减少。这意味着缓冲区在运行时能够更快达到“热”状态，即缓冲区中的数据能够更好地反映程序的访问模式。
- 但是也要注意：
    - 资源限制：虽然增加 BUFFSIZE 可以提高性能，但实际应用中要考虑到物理内存的限制。缓冲区不能无限制增大，否则可能会占用过多的内存资源，影响系统的其他部分，或者导致缓冲区管理的开销增大。
    - 递减效益：随着 BUFFSIZE 的增加，每增加一个单位大小所带来的性能提升会逐渐减少（即递减效益）。在某一点之后，增加缓冲区大小可能不会显著提高性能，因此要找到一个平衡点，既能利用缓冲区带来的性能优势，又不会无谓地消耗过多资源。
- 综上所述，合理配置缓冲区大小是优化数据库性能的关键。通过实验和监控，可以找到最适合特定应用和硬件环境的 BUFFSIZE。