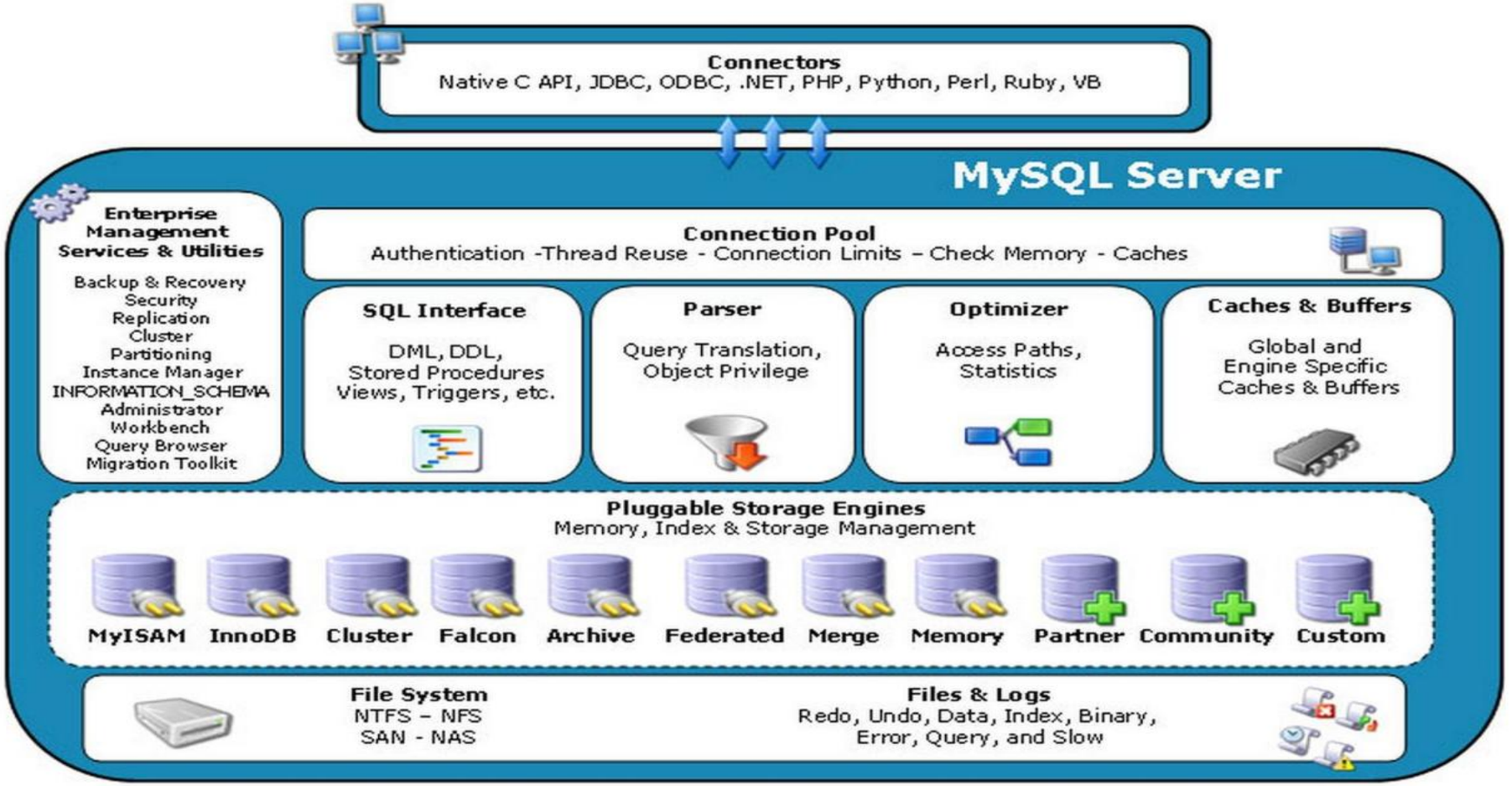
Mysql-性能优化

- mysql架构与存储引擎
- mysql底层B+tree索引机制
- mysql执行计划
- 事务隔离级别、锁、MVCC
- mysql主从复制、读写分离
- mysql分库分表

Mysql逻辑架构



问题：

- 索引是mysql实现的吗？

索引优化面试题

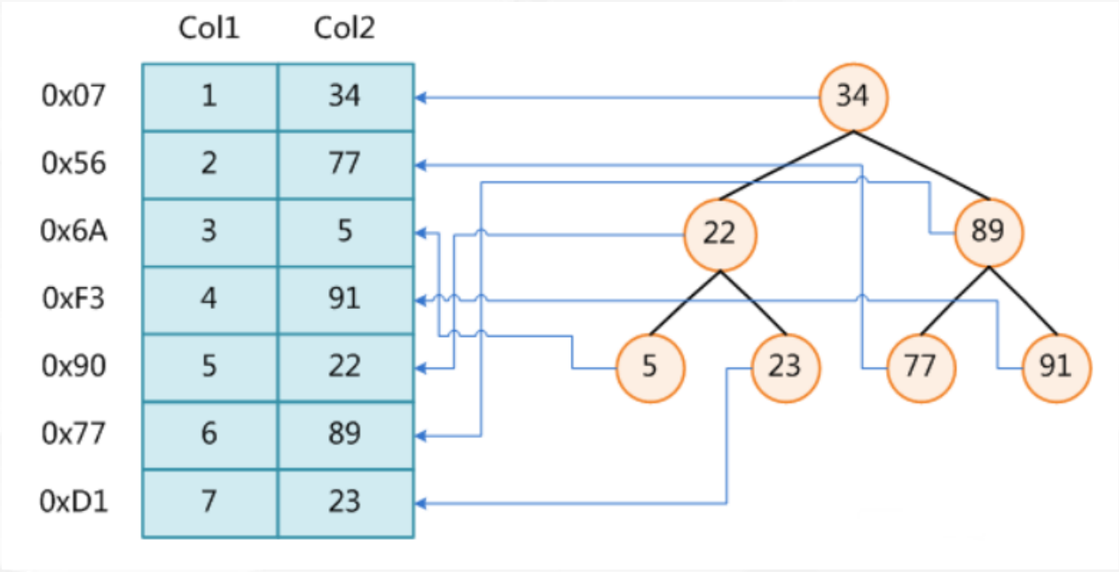
分析以下几条sql的索引使用情况

1. SELECT * FROM titles WHERE emp_no='10001' AND title='Senior Engineer' AND from_date='1986-06-26';
2. SELECT * FROM titles WHERE title='Senior Engineer' ;
3. SELECT * FROM titles WHERE emp_no > '10001';
4. SELECT * FROM titles WHERE emp_no > '10001' and title='Senior Engineer';
5. SELECT * FROM titles WHERE emp_no > '10001' order BY title;

```
SHOW INDEX FROM employees.titles;
+--------+------------+----------+--------------+-------------+-----------+-------------+------+------------+
| Table  | Non_unique | Key_name | Seq_in_index | Column_name | Collation | Cardinality | Null | Index_type |
+--------+------------+----------+--------------+-------------+-----------+-------------+------+------------+
| titles |          0 | PRIMARY  |            1 | emp_no      | A         |        NULL |      | BTREE      |
| titles |          0 | PRIMARY  |            2 | title       | A         |        NULL |      | BTREE      |
| titles |          0 | PRIMARY  |            3 | from_date   | A         |      443308 |      | BTREE      |
```

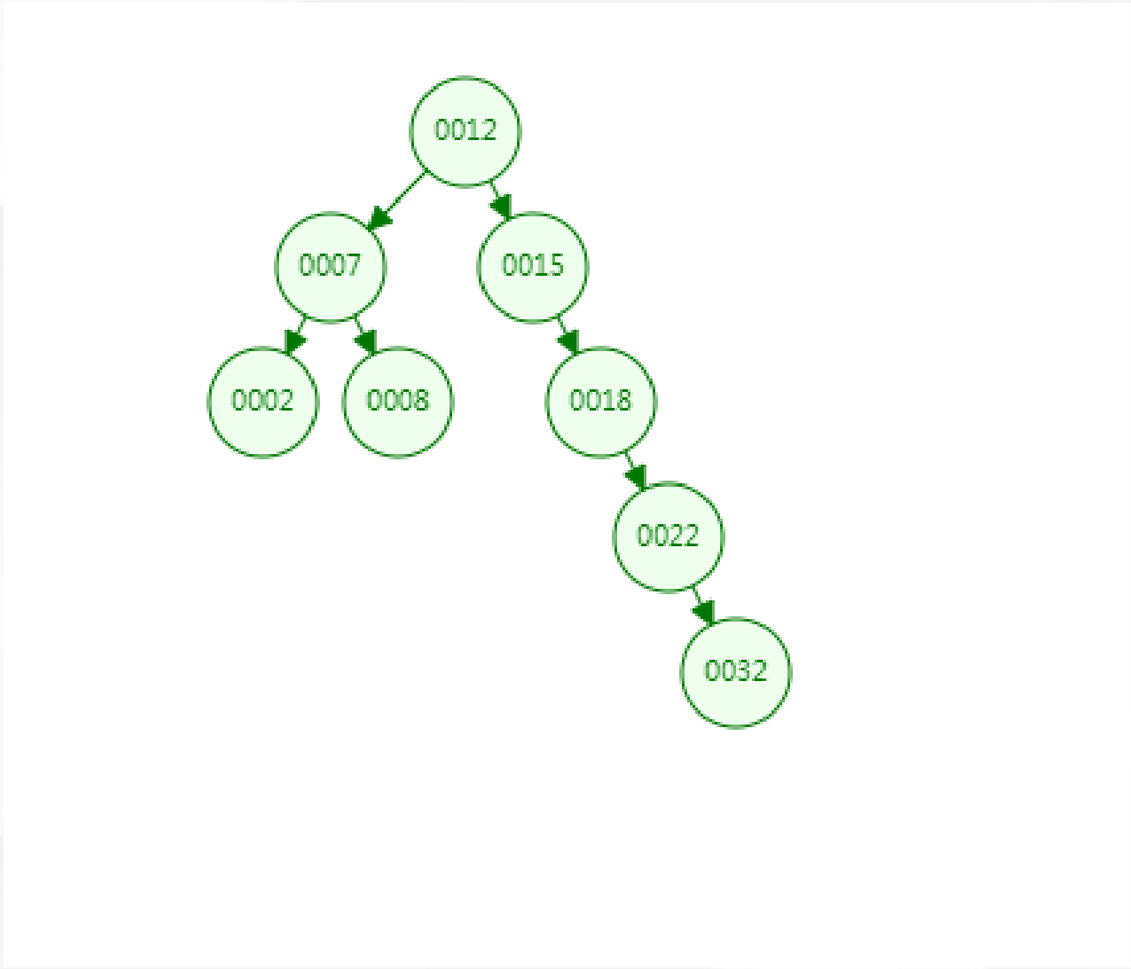
索引是什么

- 索引是帮助MySQL高效获取数据的排好序的数据结构
- 索引存储在文件里
- 索引结构
  - 二叉树
  - 红黑树
  - HASH
  - BTREE

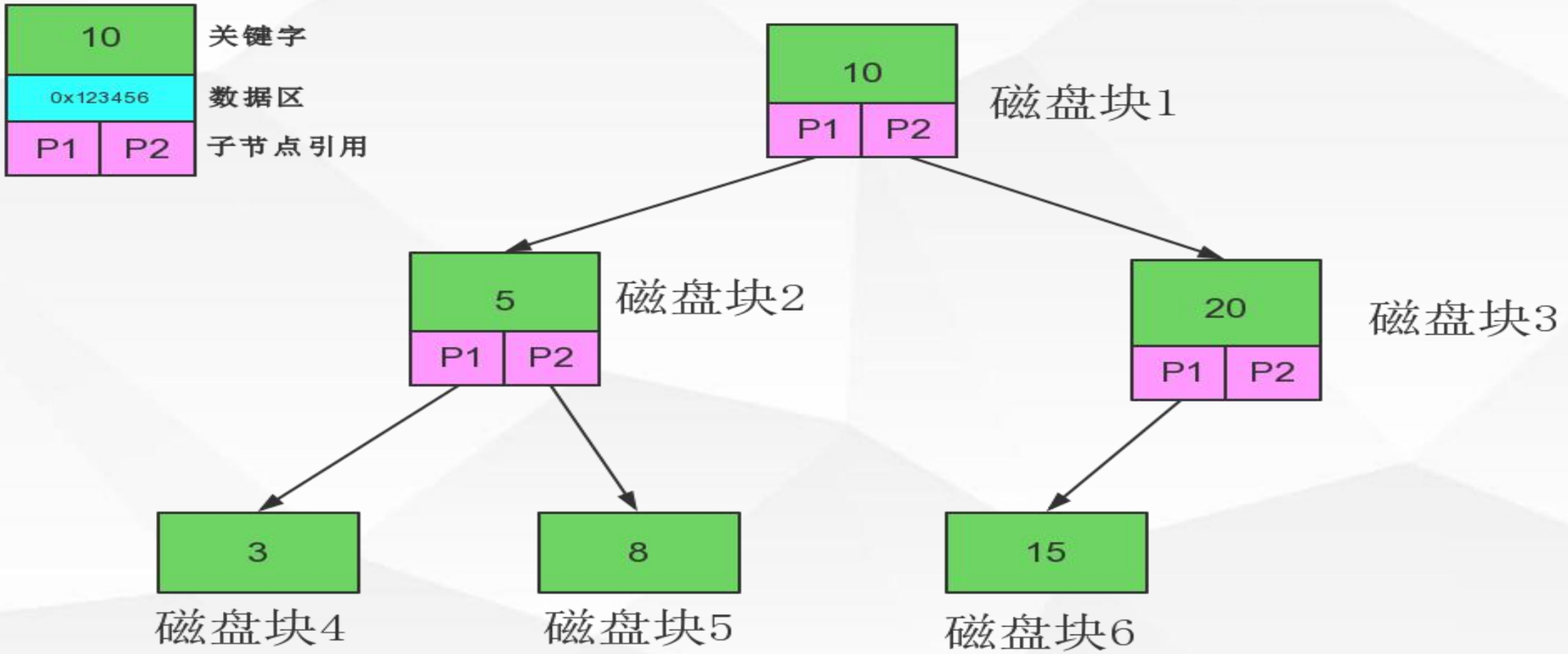


- 数据结构教学网站：https://www.cs.usfca.edu/~galles/visualization/Algorithms.html

二叉查找树



平衡二叉查找树

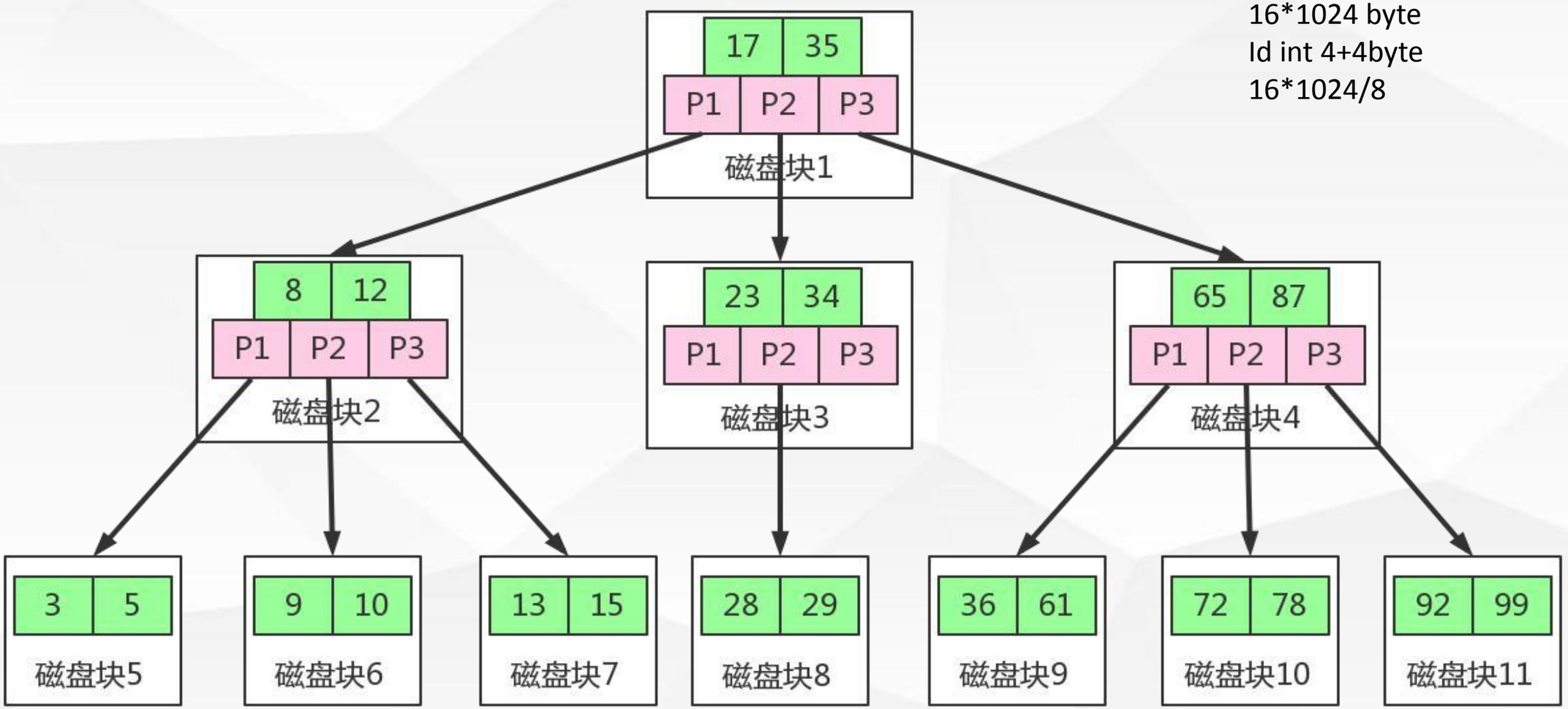

10
关键字
0x123456
数据区
P1
P2
子节点引用
10
P1
P2
磁盘块1
5
P1
P2
磁盘块2
20
P1
P2
磁盘块3
3
磁盘块4
8
磁盘块5
15
磁盘块6

有什么问题？

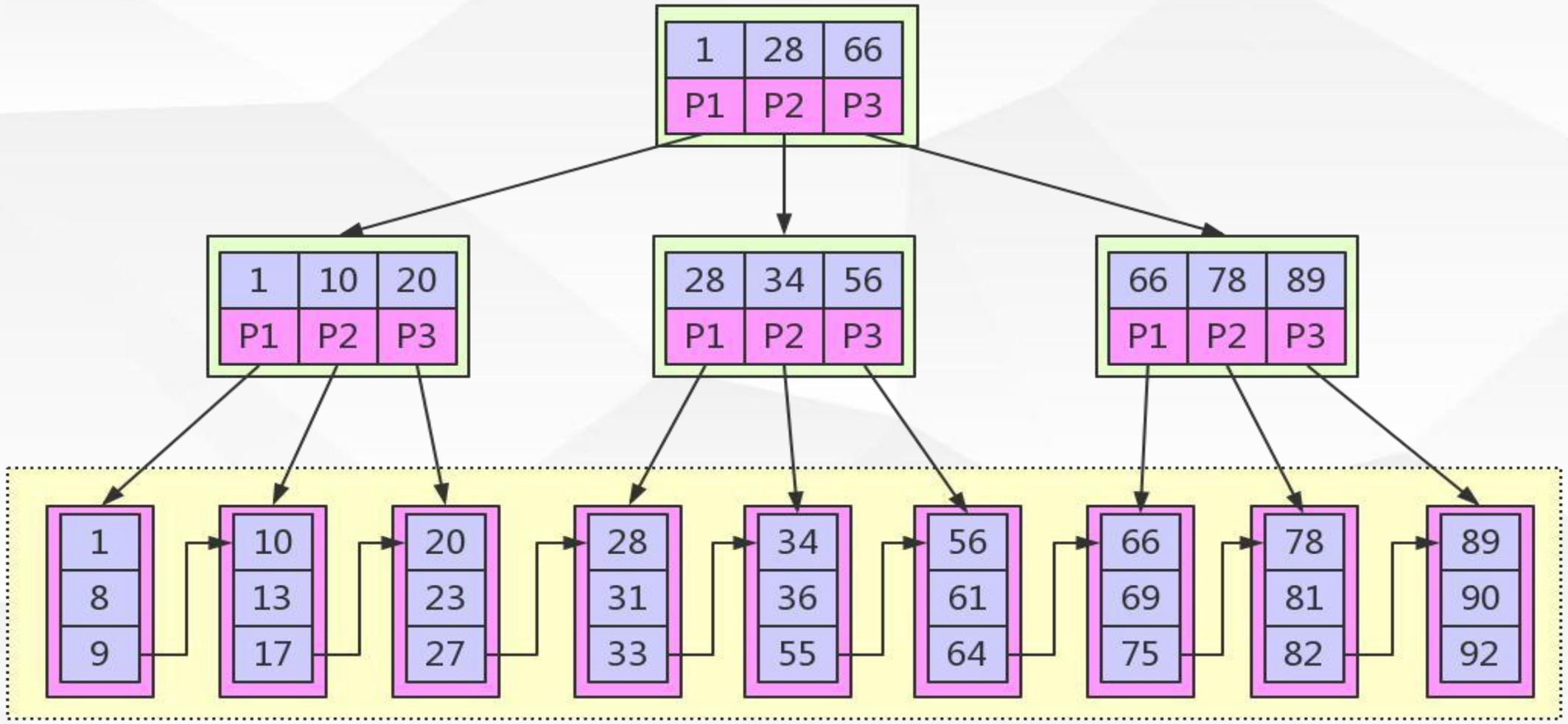
- 太深了
  - 一般使用磁盘I/O次数评价索引结构的优劣
  - 数据处的（高）深度决定着他的IO操作次数，IO操作耗时大
- 太小了
  - 每一个磁盘块（节点/页）保存的数据量太小了
  - 没有很好的利用操作磁盘IO的数据交换特性。
  - 预读：磁盘一般会顺序向后读取一定长度的数据(页的整数倍)放入内存
  - 局部性原理：当一个数据被用到时，其附近的数据也通常会马上被使用
- 正常情况
  - 树高度一般不会超过100，（一般为3到5之间）
  - 节点的大小设为等于一个页，每次新建节点直接申请一个页的空间，这样就保证一个节点物理上也存储在一个页里，就实现了一个节点的载入只需一次I/O

多路平衡查找树 B-Tree

多少路？
16*1024 byte
Id int 4+4byte
16*1024/8



多路平衡查找树 B+Tree



B-Tree和B+Tree区别

- B+节点关键字搜索采用闭合区间（设计初衷）
- B+非叶节点不保存数据相关信息，只保存关键字和子节点的引用
- B+关键字对应的数据保存在叶子节点中
- B+叶子节点是顺序排列的，并且相邻节点具有顺序引用的关系

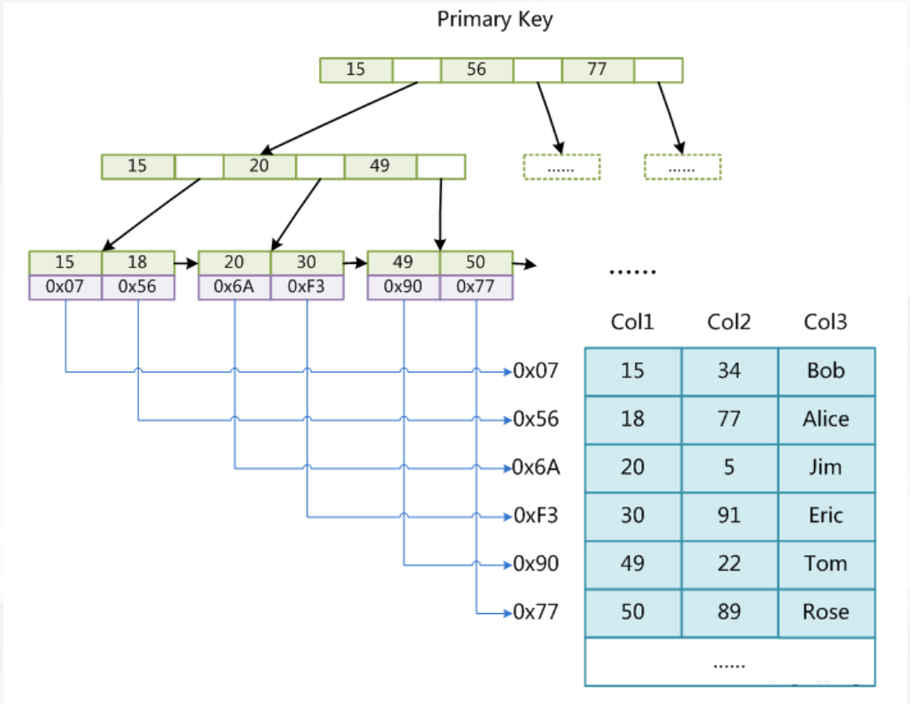
为什么选用B+Tree？

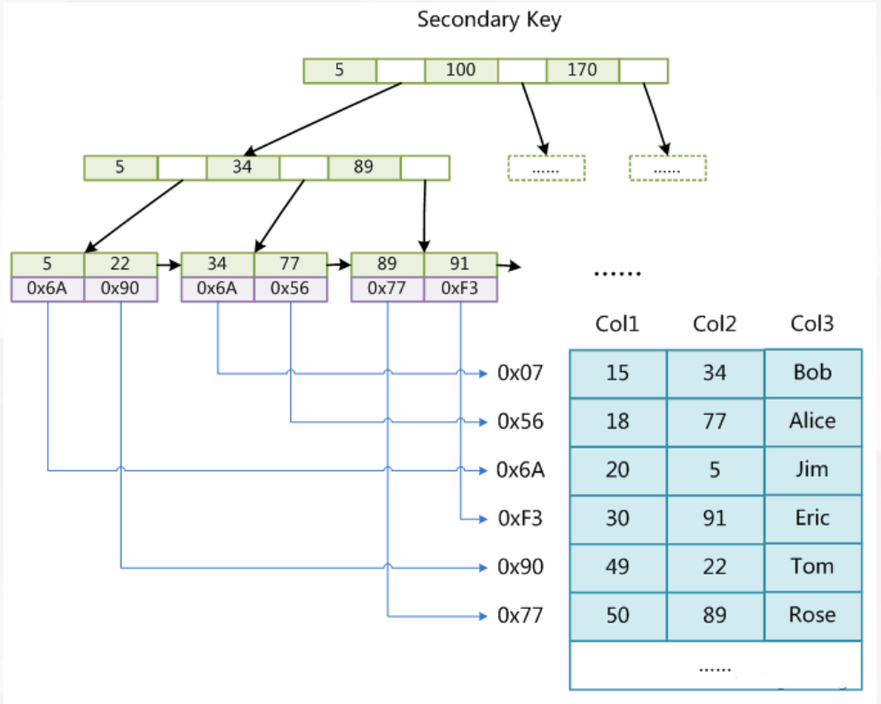
- 每个节点可以容纳更多的关键字
- B+tree减少磁盘的访问次数
- 访问磁盘一次的时间比内存上千次的比较的时间更多，所以B+tree性能更高
- 数据指针连接在一起，方便顺序遍历
- B+tree的查询效率更加稳定
    - 所有关键字查询路径长度一样，每一个查询效率相当。

索引底层数据结构与算法

- MyISAM索引实现(非聚集)
  - MyISAM索引文件和数据文件是分离的

```
user_myisam.frm
user_myisam.MYD
user_myisam.MYI
```

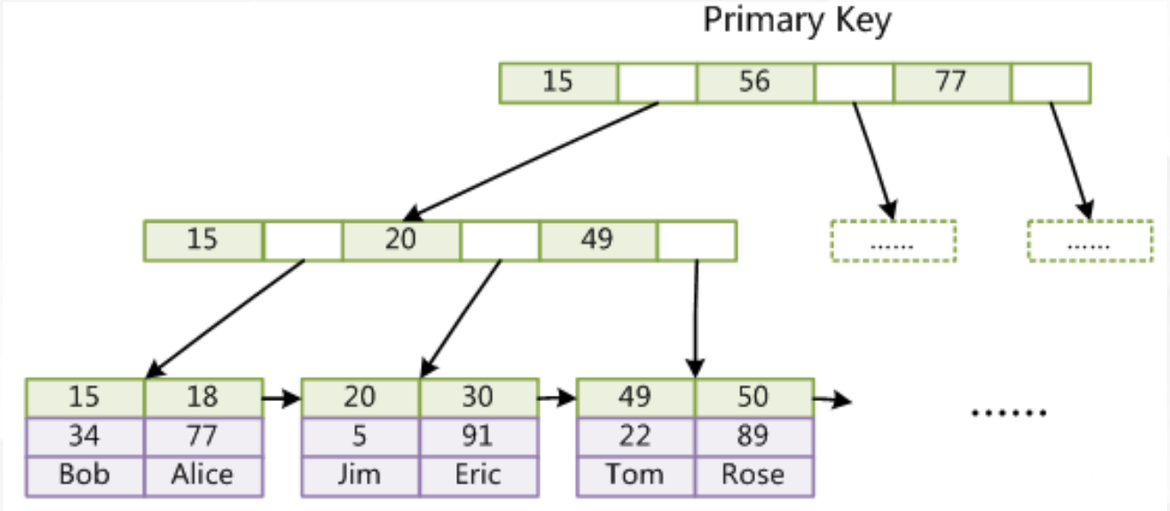


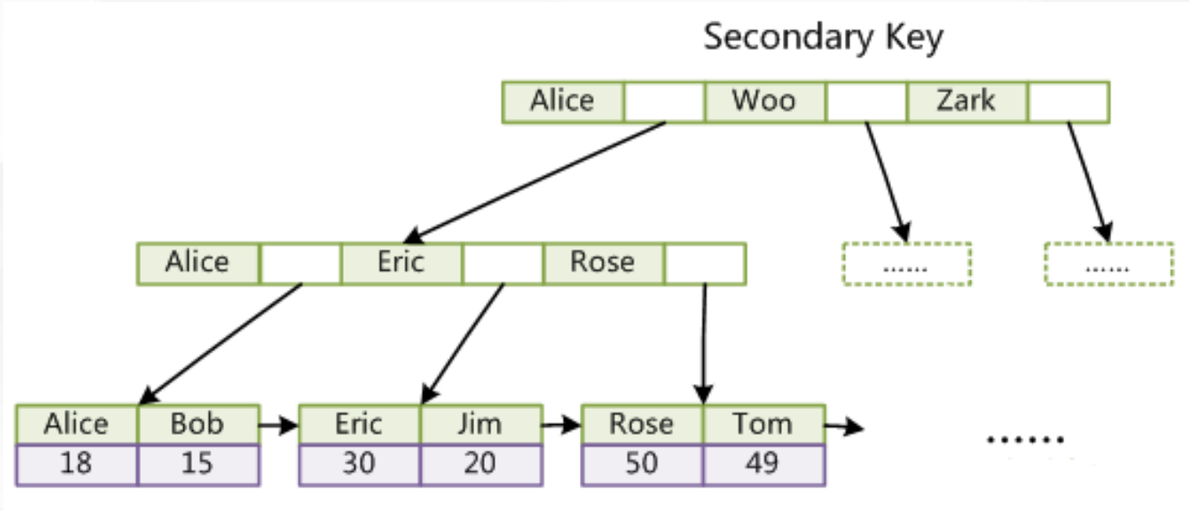


索引底层数据结构与算法

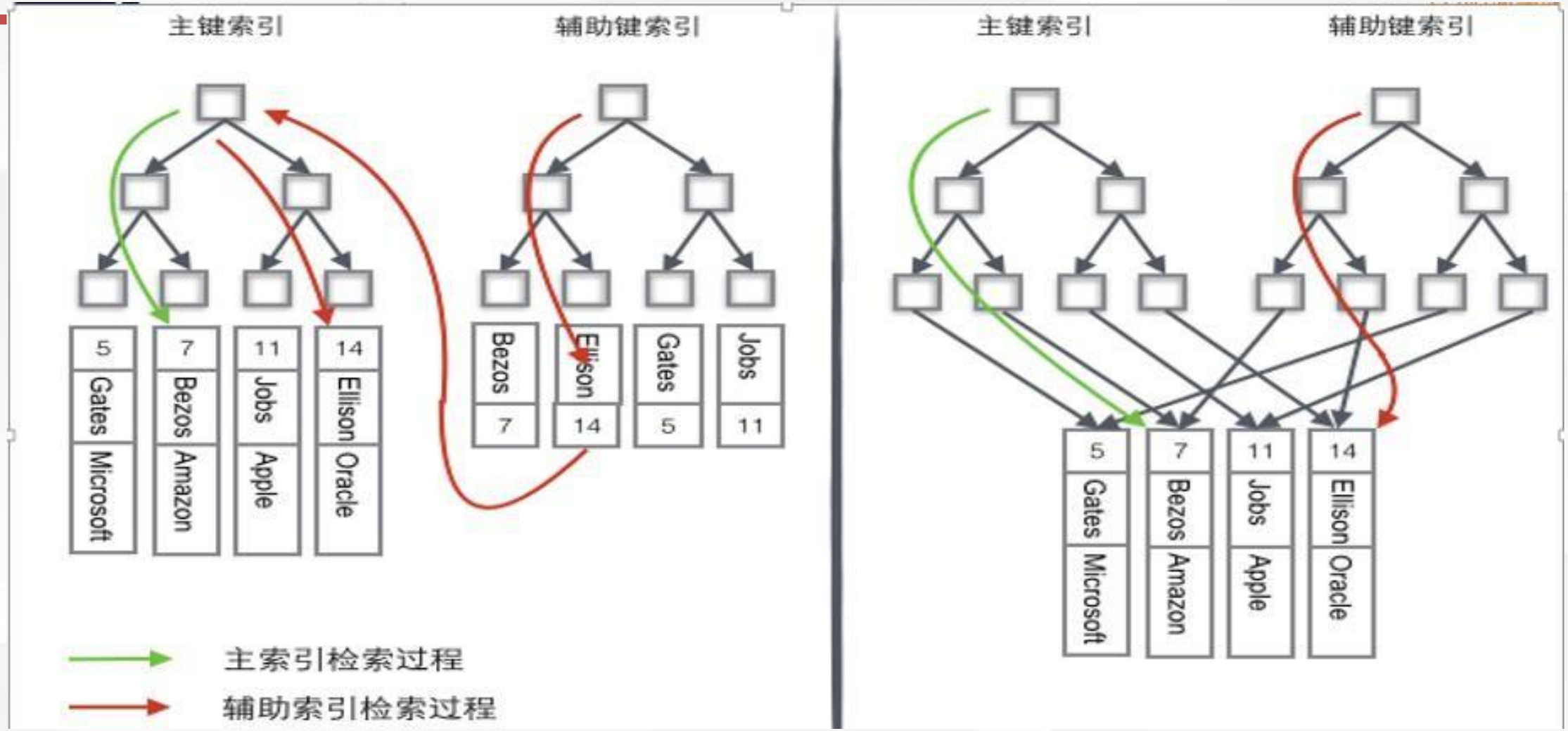
- InnoDB索引实现(聚集)
  - 数据文件本身就是索引文件
  - 表数据文件本身就是按B+Tree组织的一个索引结构文件
  - 聚集索引-叶节点包含了完整的数据记录
  - 为什么InnoDB表必须有主键，并且推荐使用整型的自增主键？（没有主键隐式ID）
  - 为什么非主键索引结构叶子节点存储的是主键值？(一致性和节省存储空间)

```
user_innodb.frm
user_innodb.ibd
```

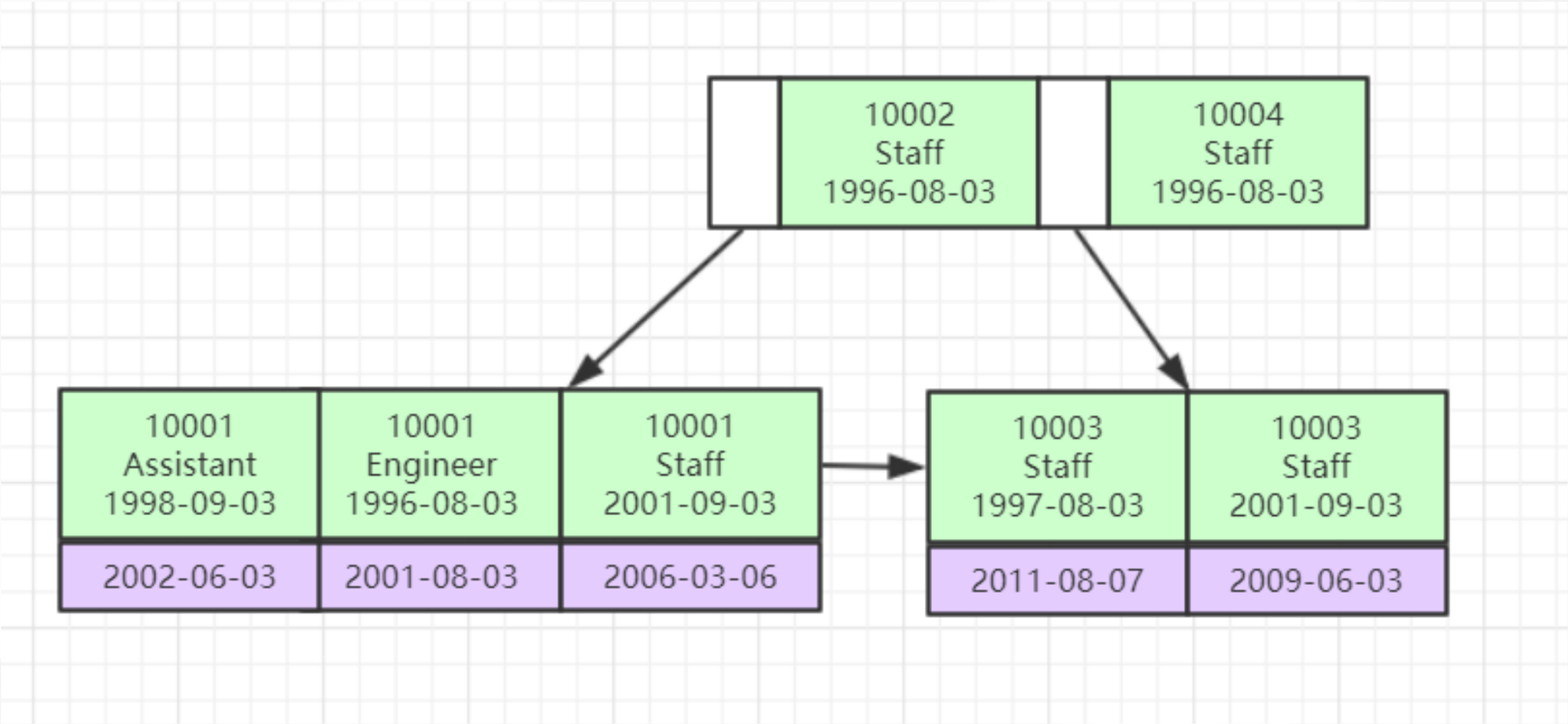




Innodb vs Myisam



联合索引结构


10002
Staff
1996-08-03
10004
Staff
1996-08-03
10001
Assistant
1998-09-03
10001
Engineer
1996-08-03
10001
Staff
2001-09-03
10003
Staff
1997-08-03
10003
Staff
2001-09-03
2002-06-03
2001-08-03
2006-03-06
2011-08-07
2009-06-03

如何选择索引列

- 索引选择性=不重复的索引值/数据表的记录数
- 使用最频繁的列优先【最左匹配原则】
  - sex?
- 将选择性最高的列优先【离散度高原则】
- 宽度小的列优先【最少空间原则】

联合索引如何选择列-面试题

- Select * from payment where staff_id=2 and customer_id=584
  - (staff_id, customer_id) 索引顺序如何选择?
  - Select count(distinct staff_id)/count(*), count(distinct customer_id)/count(*), count(*)
    from payment
    - 0.0001, 0.0373, 16049

联合索引如何选择列-面试题

```
mysql> SELECT COUNT(DISTINCT threadId) AS COUNT_VALUE
    -> FROM Message
    -> WHERE (groupId = 10137) AND (userId = 1288826) AND (anonymous = 0)
    -> ORDER BY priority DESC, modifiedDate DESC
```

```
           id: 1
  select_type: SIMPLE
        table: Message
         type: ref
          key: ix_groupId_userId
      key_len: 18
          ref: const,const
         rows: 1251162
        Extra: Using where
```

```
mysql> SELECT COUNT(*), SUM(groupId = 10137),
    -> SUM(userId = 1288826), SUM(anonymous = 0)
    -> FROM Message\G
*************************** 1. row ***************************
                count(*): 4142217
 sum(groupId = 10137): 4092654
sum(userId = 1288826): 1288496
   sum(anonymous = 0): 4141934
```

覆盖索引

如果查询列可通过索引节点中的关键字直接返回，则该索引称之为 覆盖索引。

```
mysql> EXPLAIN SELECT * FROM products WHERE actor='SEAN CARREY'
    -> AND title like '%APOLLO%'\G

mysql> EXPLAIN SELECT *
    -> FROM products
    ->    JOIN (
    ->       SELECT prod_id
    ->       FROM products
    ->       WHERE actor='SEAN CARREY' AND title LIKE '%APOLLO%'
    ->    ) AS t1 ON (t1.prod_id=products.prod_id)\G
```

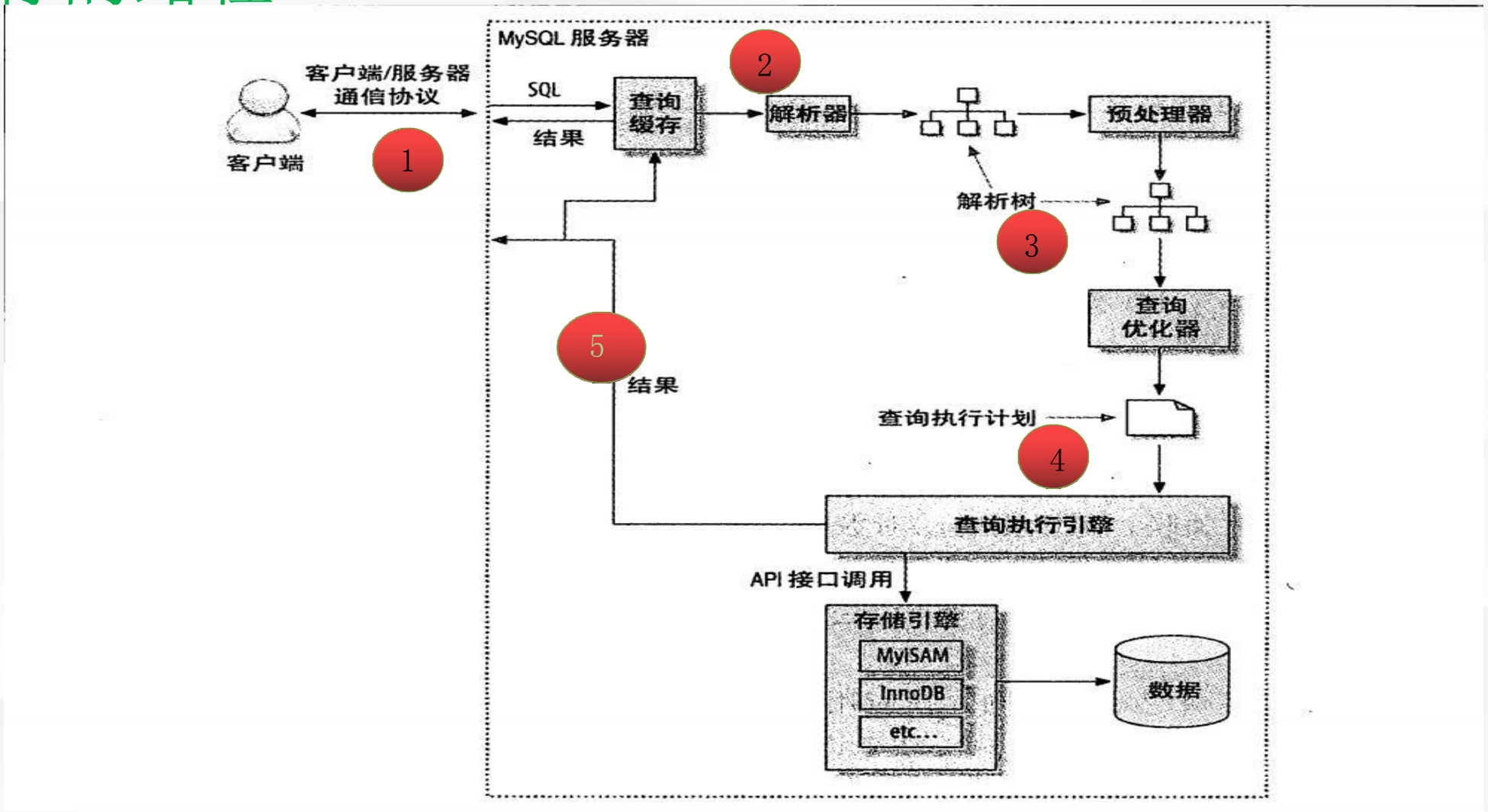
- SEAN CARREY 出演了30000部作品，其中有20000部包含了APLLO
- SEAN CARREY 出演了30000部作品，其中有40部包含了APLLO
- SEAN CARREY 出演了50部作品，其中有10部包含了APLLO

- 上面那种数据集使用覆盖索引后性能上表现最好？

第 2 部分

执行计划

查询执行的路径


MySQL 服务器
客户端/服务器
通信协议
客户端
SQL
结果
1
查询
缓存
2
解析器
预处理器
解析树
3
查询
优化器
5
结果
查询执行计划
4
查询执行引擎
API 接口调用
存储引擎
MyISAM
InnoDB
etc...
数据

执行计划

- **使用EXPLAIN关键字可以模拟优化器执行SQL语句，从而知道MySQL是 如何处理你的SQL语句的。分析你的查询语句或是结构的性能瓶颈**
- 案例分析
- 58同城30条军规

第 3 部分

存储引擎

存储引擎介绍

- 插拔式的插件方式
- 存储引擎是指定在表之上的，即一个库中的每一个表都可 以指定专用的存储引擎。
- 不管表采用什么样的存储引擎，都会在数据区，产生对应 的一个frm文件（表结构定义描述文件）
- show variables like '%storage_engine%';

存储引擎- CVS

- 数据存储以**CSV**文件
- 特点：

  不能定义没有索引、列定义必须为NOT NULL、不能设置自增列
  -->不适用大表或者数据的在线处理

  CSV数据的存储用,隔开，可直接编辑CSV文件进行数据的编排
  -->数据安全性低

  注：编辑之后，要生效使用flush table XXX 命令

- 应用场景：

  数据的快速导出导入 表格直接转换**CSV**

存储引擎- Memory

- 文件系统存储特定，也称为heap存储引擎，数据保存在内存中
- 特点:
  - 支持hash索引和Btree索引 默认hash(O(1))
  - 所有字段都固定长度varchar(10)=char(100)
  - 不支持Blog和Text等大字段
  - 表级锁
  - 最大大小由max_heap_table_size参数决定
- 使用场景
  - Hash索引用于查找或者映射表（邮编和地区对应表）
  - 保存数据分析中产生的中间表
  - 数据容易丢失，要求数据可再生

存储引擎- Memory


Memory存储引擎表
VS
临时表
系统使用临时表
超过限制使用Myisam临时表
未超限制使用Memory表
create temporary teble 建立的临时表

存储引擎- Myisam

Mysql5.5版本之前的默认存储引擎较多的系统表也还是使用这个存储引擎 系统临时表也会用到**Myisam**存储引擎

特点:

- select count(*) from table 无需进行数据的扫描
- 数据（MYD）和索引（MYI）分开存储
- 表级锁
- 不支持事务

- 使用场景
  - 非事务形应用（数据仓库，报表，日志数据）
  - 只读类应用
  - 空间类应用（gis、地图、空间函数、坐标）

存储引擎- Innodb

- **Mysql5.5**及以后版本的默认存储引擎
- innodb_file_per_table
  - ON独立表空间（.ibd）
  - OFF系统表空间（ibdata1）
  - 特性：
    - 系统表空间无法简单收缩文件大小
    - 独立表空间可以通过optimize table收缩系统文件
    - 系统表空间会产生IO瓶颈
    - 独立表空间可以同时向多个文件刷新数据
- 建议:
  - Innodb使用独立表空间

存储引擎- Innodb

- 特性
  - Its DML operations follow the ACID model [事务ACID]
  - Row-level locking[行级锁] （并发程度高）
  - InnoDB tables arrange your data on disk to optimize queries based on primary keys[聚集索引（主键索引）方式进行数据存储]
  - To maintain data integrity, InnoDB supports FOREIGN KEY constraints[支持外键关系保证数据完整性]
  - Undo Log 和Redo Log

https://dev.mysql.com/doc/refman/5.7/en/innodb-introduction.html

存储引擎

对比项	MyISAM	InnoDB
主外键	不支持	支持
事务	不支持	支持
行表锁	表锁，即使操作一条记录也会锁住整个表 不适合高并发的操作	行锁,操作时只锁某一行，　不对其它行有影响 适合高并发的操作
缓存	只缓存索引，不缓存真实数据	不仅缓存索引还要缓存真实数据，对内存要求较高，而且内存大小对性能有决定性的影响
表空间	小	大
关注点	性能	事务
默认安装	Y	Y

第 4 部分

事务、锁、MVCC

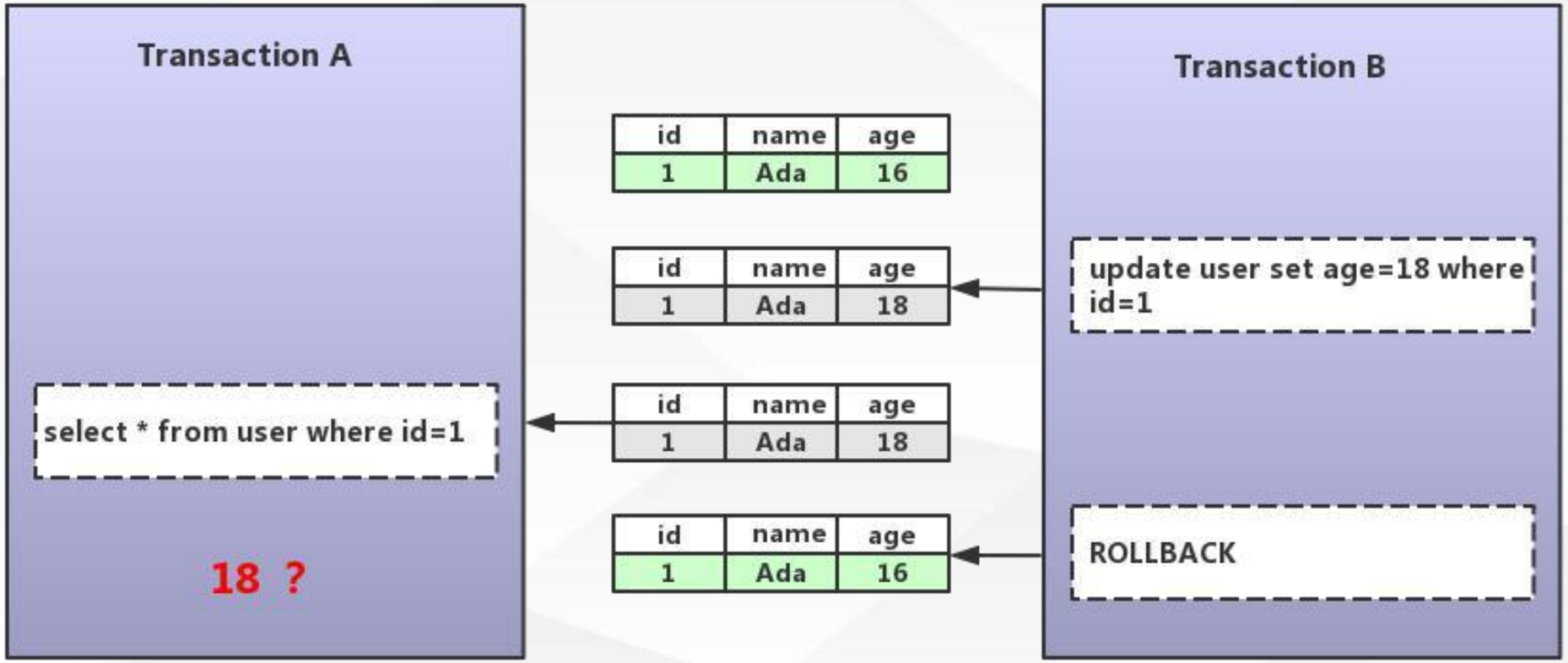
什么是事务

- 事务：
  数据库操作的最小工作单元，是作为单个逻辑工作单元执行的一系列操作； 事务是一组不可再分割的操作集合（工作逻辑单元）
- 典型的事务场景（转账）
  - update account set balance = balance - 1000 where userID = 2;
  - update account set balance = balance +1000 where userID = 1;
- Mysql中开启事务：
  - begin /start transaction -- 手工开启事务
  - commit / rollback -- 事务提交或回滚
  - set session autocommit = on/off; ---设定事务是否自动开启

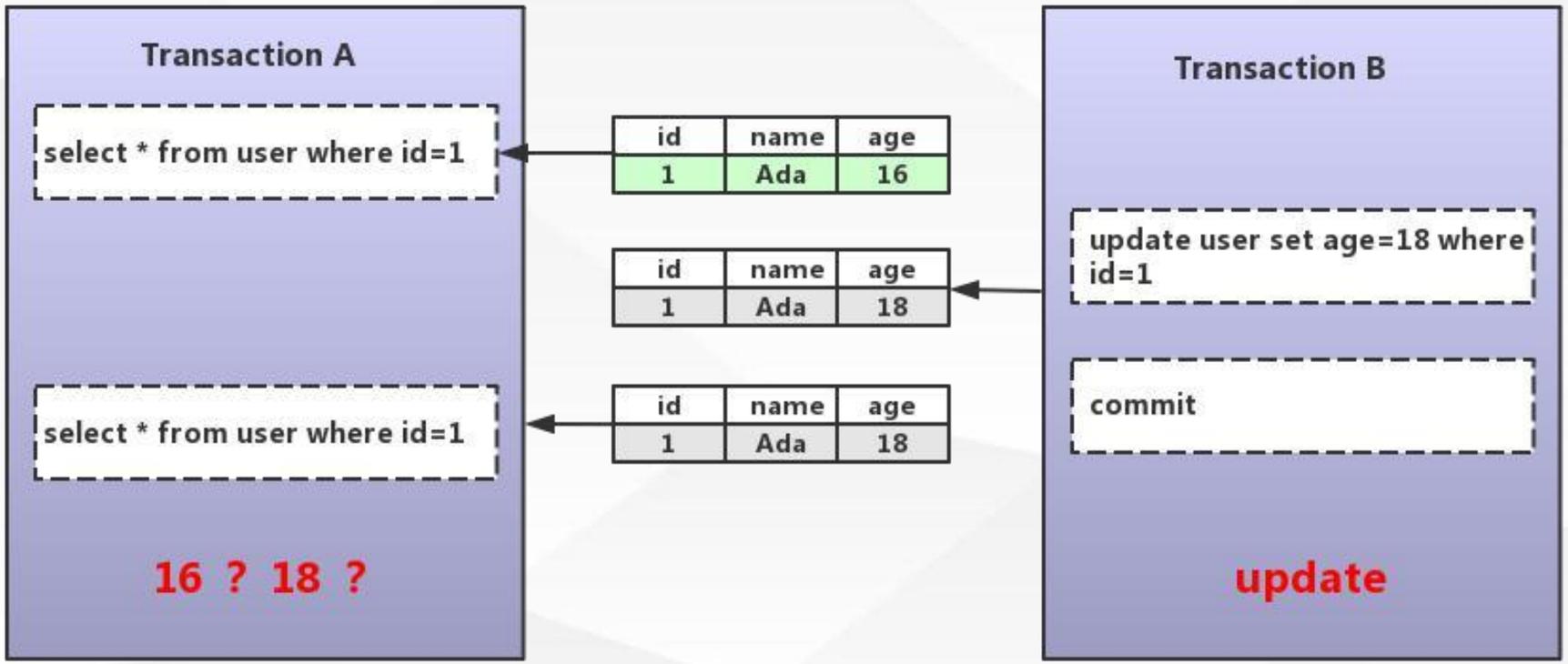
事务的ACID特性

- 原子性（Atomicity）
  - 最小的工作单元，整个工作单元要么一起提交成功，要么全部失败回滚
- 一致性（Consistency）
  - 事务中操作的数据及状态改变是一致的，即写入资料的结果必须完全符合预设的规则， 不会因为出现系统意外等原因导致状态的不一致
- 隔离性（Isolation）
  - 一个事务所操作的数据在提交之前，对其他事务的可见性设定（一般设定为不可见）
- 持久性（Durability）
  - 事务所做的修改就会永久保存，不会因为系统意外导致数据的丢失

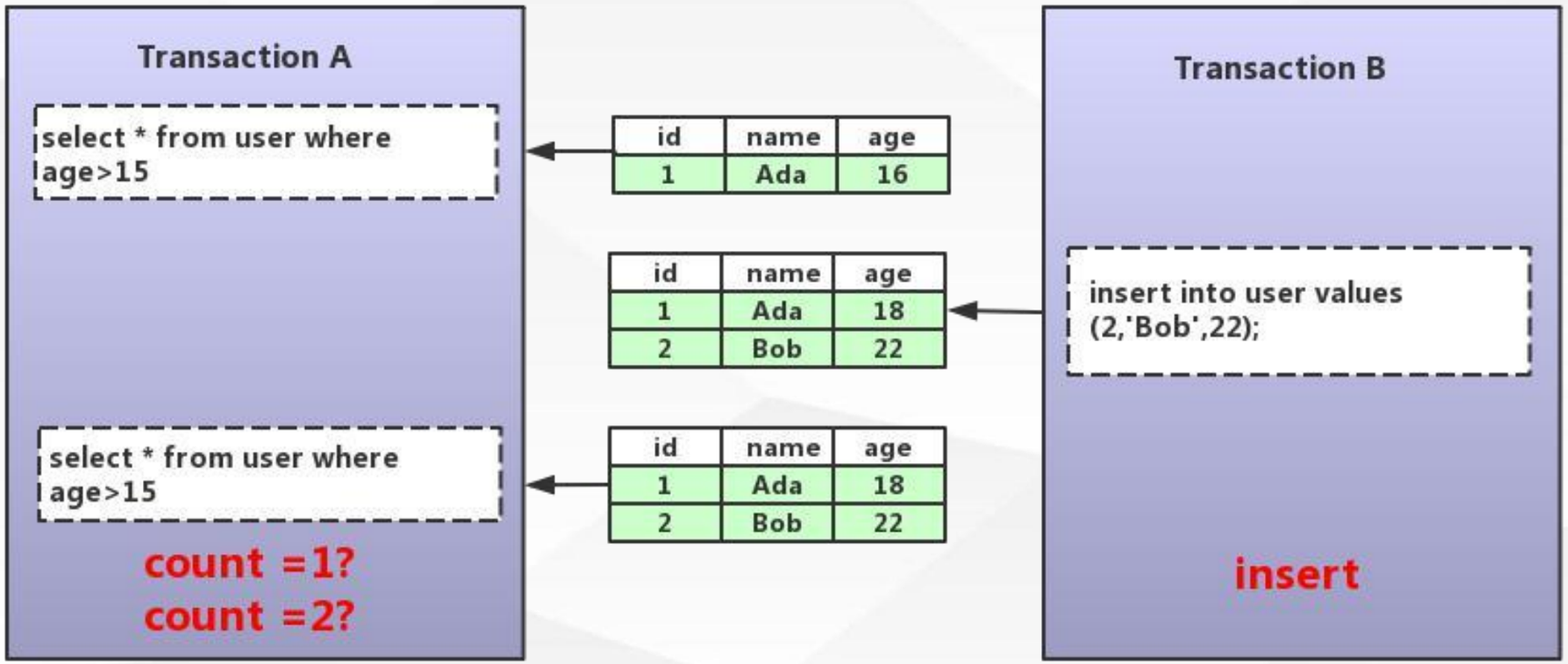
事务并发带来哪些问题


Transaction A
select * from user where id=1
18 ?
id name age
1 Ada 16
id name age
1 Ada 18
id name age
1 Ada 18
id name age
1 Ada 16
Transaction B
update user set age=18 where id=1
ROLLBACK

事务并发带来哪些问题


Transaction A
select * from user where id=1
select * from user where id=1
16 ? 18 ?
id name age
1 Ada 16
id name age
1 Ada 18
id name age
1 Ada 18
Transaction B
update user set age=18 where id=1
commit
update

事务并发带来哪些问题


Transaction A
select * from user where age>15
select * from user where age>15
count =1?
count =2?
id name age
1 Ada 16
id name age
1 Ada 18
2 Bob 22
id name age
1 Ada 18
2 Bob 22
Transaction B
insert into user values (2,'Bob',22);
insert

事务4种隔离级别

- Read Uncommitted（未提交读） --未解决并发问题
  - 事务未提交对其他事务也是可见的，脏读（dirty read）
- Read Committed（提交读） --解决脏读问题
  - 一个事务开始之后，只能看到自己提交的事务所做的修改，不可重复读（nonrepeatableread）
- Repeatable Read （可重复读） --解决不可重复读问题
  - 在同一个事务中多次读取同样的数据结果是一样的，这种隔离级别未定义解决幻读的问题
- Serializable（串行化） --解决所有问题 最高的隔离级别

Innodb引擎对隔离级别的支持程度

事务隔离级别	脏读	不可重复读	幻读
未提交读（Read Uncommitted）	可能	可能	可能
已提交读（Read Committed）	不可能	可能	可能
可重复读（Repeatable Read）	不可能	不可能	对 InnoDB 不可能
串行化（Serializable）	不可能	不可能	不可能

隔离级别到底如何实现的呢？ → 锁、MVCC

Innodb锁的类型

- 共享锁（行锁）：Shared Locks
- 排它锁（行锁）：Exclusive Locks
- 意向锁共享锁（表锁）：Intention Shared Locks
- 意向锁排它锁（表锁）：Intention Exclusive Locks
- 自增锁：AUTO-INC Locks
- 记录锁 Record Locks
- 间隙锁 Gap Locks
- 临键锁 Next-key Locks

https://dev.mysql.com/doc/refman/5.7/en/innodb-locking.html

共享锁(Shared Locks) vs 排它锁(Exclusive Locks)

- 共享锁：

  又称为读锁，简称S锁，顾名思义，共享锁就是多个事务对同一数据可以共享一把锁，都能访问到数据，但是只能读不能修改。

  - 加锁方式：

    ```
    select * from users WHERE id=1 LOCK IN SHARE MODE;

    commit/rollback
    ```

- 排他锁：

  又称为写锁，简称X锁，排他锁不能与其他锁并存，如果一个事务获取一个数据行的排他锁，其它事务就不能再获取该行的锁（共享锁、排它锁），只有该获取了排他锁的事务是可以对 数据行进行读取和修改，（其他事务要读取数据来自于快照）

  - 加锁方式：

    ```
    delete / update / insert 默认加上X锁
    SELECT * FROM table_name WHERE ... FOR UPDATE
    commit/rollback
    ```

Innodb行锁

- InnoDB的行锁是通过给索引上的索引项加锁来实现的。
- 只有通过索引条件进行数据检索，InnoDB才使用行级锁，否则，InnoDB 将使用表锁（锁住索引的所有记录）
- 表锁：lock tables xx read/write;

意向共享锁（IS）&意向排它锁（IX）

- 意向共享锁(IS)
  - 表示事务准备给数据行加入共享锁，即一个数据行加共享锁前必须先取得该表的IS锁， 意向共享锁之间是可以相互兼容的
- 意向排它锁(IX)
  - 表示事务准备给数据行加入排他锁，即一个数据行加排他锁前必须先取得该表的IX锁， 意向排它锁之间是可以相互兼容的
- 意向锁(IS、IX)是InnoDB数据操作之前自动加的，不需要用户干预
- 意义：
  - 当事务想去进行锁表时，可以先判断意向锁是否存在，存在时则可快速返回该表不能启用表锁

自增锁AUTO-INC Locks

- 针对自增列自增长的一个特殊的表级别锁
- `show variables like 'innodb_autoinc_lock_mode';`
- **innodb_autoinc_lock_mode:**
  - 0: 表锁，且必须等待当前SQL执行完成后或者回滚掉才会释放
  - **1：** 轻量的锁，立即获得该锁，并立即释放，而不必等待当前SQL执行完成
  - 2：所有insert情况下表获得最大并发度，主从复制时，可能出现主键冲突
- 默认取值1，代表连续，事务未提交ID永久丢失

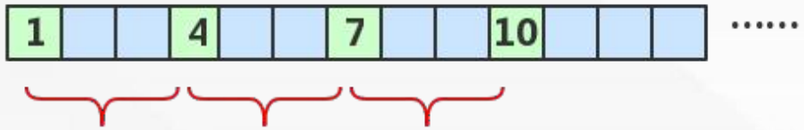
记录锁(Record)&间隙锁(Gap)&临键锁(Next-key)

- Next-key locks：锁住记录+区间（左开右闭）
  - 当sql执行按照索引进行数据的检索时,查询条件为范围查找（between and、<、>等）并有数 据命中则此时SQL语句加上的锁为Next-key locks，锁住索引的记录+区间（左开右闭）
- Gap locks：
  - 锁住数据不存在的区间（左开右开）
    - 当sql执行按照索引进行数据的检索时，查询条件的数据不存在，这时SQL语句加上的锁即Gap locks，锁住索引不存在的区间（左开右开）
- Record locks：
  - 锁住具体的索引项
  - 当sql执行按照唯一性（Primary key、Unique key）索引进行数据的检索时，查询条件等值匹 配且查询的数据是存在，这时SQL语句加上的锁即为记录锁Record locks，锁住具体的索引项

临键锁(Next-key)

Next-Key Lock： InnoDB默认的行锁算法

(- ∞,1] (1,4] (4,7] (7,10] (10, + ∞)



Next-key Lock = Gap Lock + Record lock

```
select * from t where id>5 and id<9
for update;
```

锁住：
(4,7] (7,10]

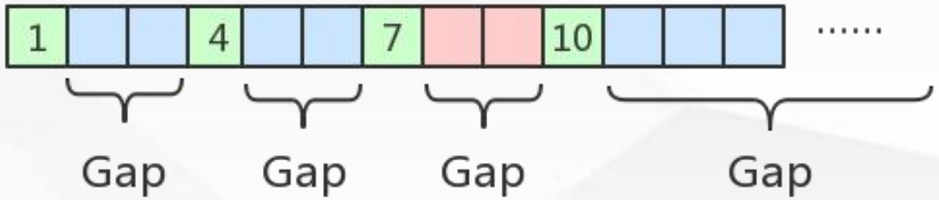
Msyql为什么采用Next-key作为行锁的默认算法？

间隙锁(Gap)

当记录不存在，临键锁退化成Gap锁

Gap Lock：范围查询或等值查询且记录不存在

(- ∞,1) (1,4) (4,7) (7,10) (10, + ∞)



Gap锁之间不冲突

```
select * from t where id >4
  and id <6 for update;
```

```
select * from t where id =6
    for update;
```

锁住：(4,7)

```
select * from t where id
  >20 for update;
```

锁住：(10,+∞)

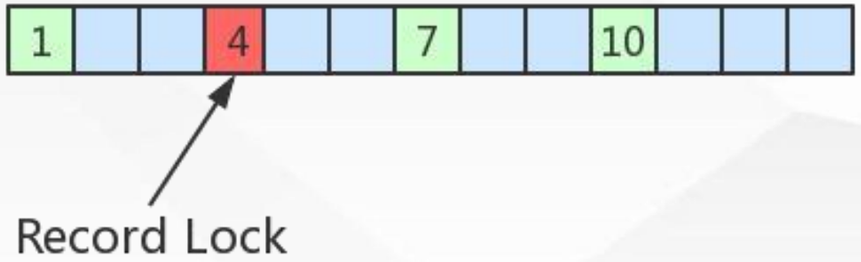
Gap只在RR事务隔离级别存在

记录锁(Gap)

唯一性（主键/唯一）索引，条件为精准匹配，退化成Record锁

Record Lock ：唯一索引 unique key(id)

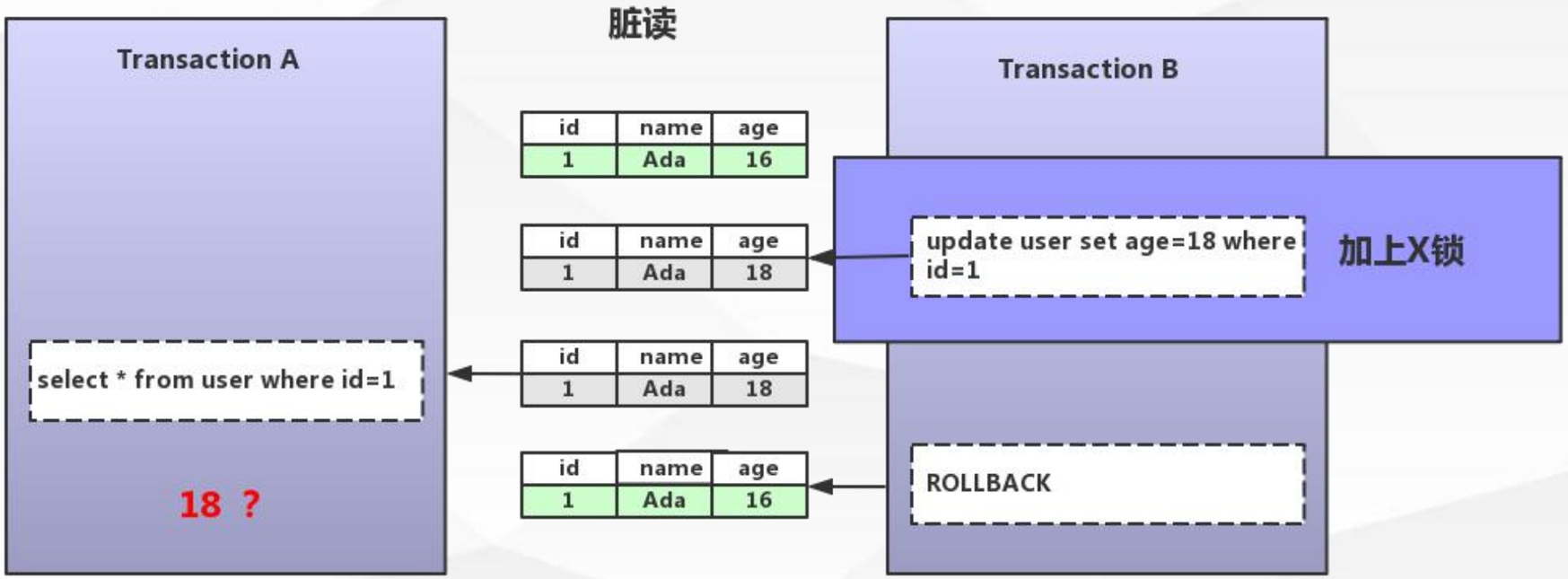
等值查询，精准匹配



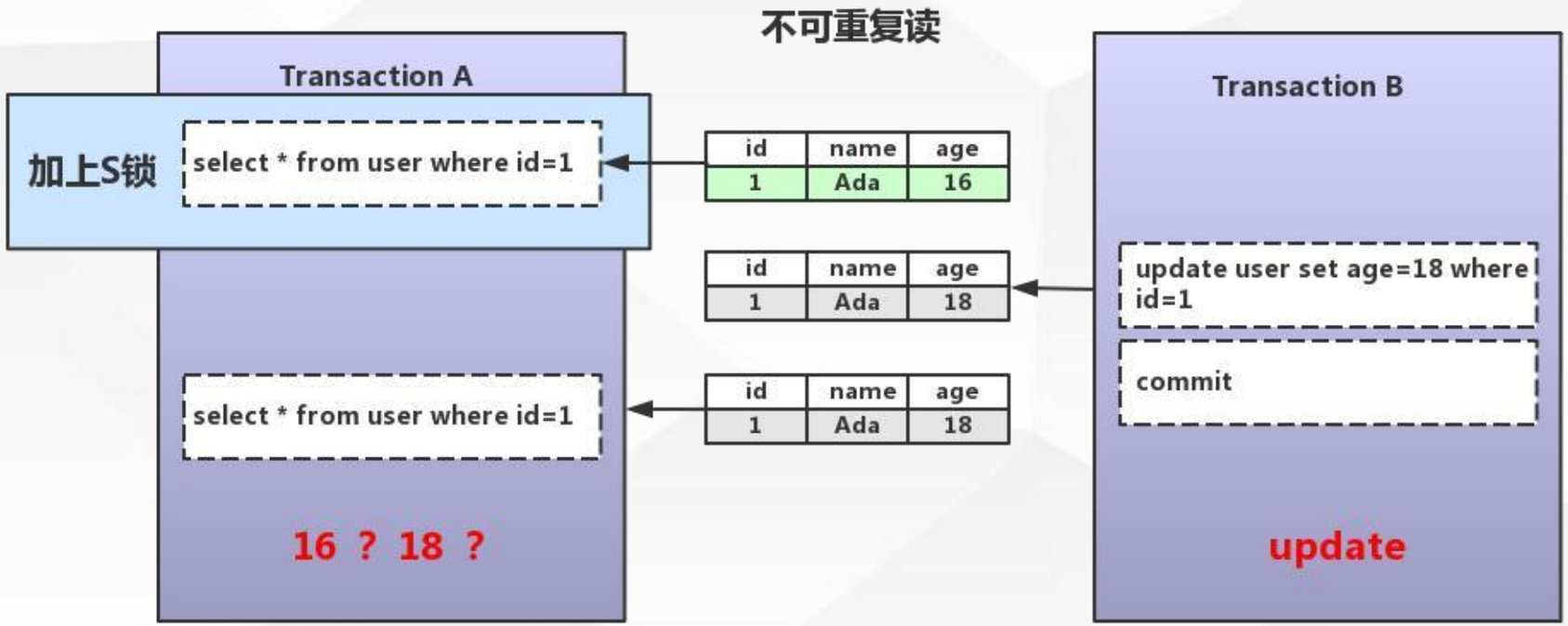

```
select * from t
where id =4 for update;
```

锁住：id=4

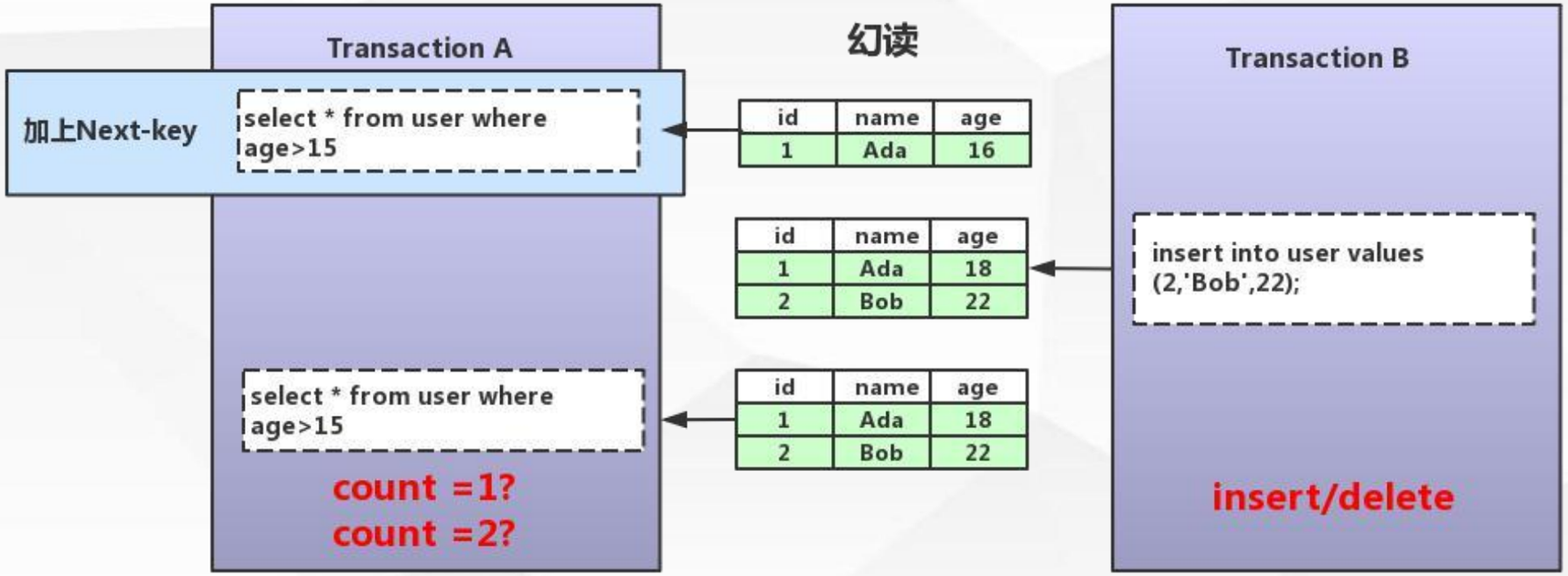
利用锁解决脏读


脏读
Transaction A
select * from user where id=1
18 ?
id name age
1 Ada 16
id name age
1 Ada 18
id name age
1 Ada 18
id name age
1 Ada 16
Transaction B
update user set age=18 where id=1
加上X锁
ROLLBACK

利用锁解决不可重复读


不可重复读
Transaction A
加上S锁
select * from user where id=1
id name age
1 Ada 16
select * from user where id=1
id name age
1 Ada 18
16 ? 18 ?
Transaction B
update user set age=18 where id=1
id name age
1 Ada 18
commit
update

利用锁解决幻读


Transaction A
加上Next-key
select * from user where age>15
幻读
id name age
1 Ada 16
id name age
1 Ada 18
2 Bob 22
select * from user where age>15
id name age
1 Ada 18
2 Bob 22
count =1?
count =2?
Transaction B
insert into user values (2,'Bob',22);
insert/delete

死锁

产生的原因：

- 当多个事务同时持有和请求同一资源上的锁而产生循环依赖的时候就产生了死锁。死锁发生在事务试图以不同的顺序锁定资源

解决办法：

- 数据库实现了各种死锁探查和超时机制。当前InnoDB处理死锁的方式是回滚持有最少排他行级锁的事务。
- 尽量避免并发的执行涉及到修改数据的语句。
- 大事务拆小。大事务更倾向于死锁，如果业务允许，将大事务拆小。
- 要求每一个事务一次就将所有要使用到的数据全部加锁，否则就不允许执行。
- 预先规定一个加锁顺序，所有的事务都必须按照这个顺序对数据执行封锁。如不同的过程在事务内部对对象的更新执行顺序应尽量保证一致。
- 相关字段建立索引。优化后的执行计划使用了索引查找，将大幅提升该查询语句的性能，降低了锁定资源的时间，同时也减少了锁定资源的范围，这样就降低了锁资源循环等待事件发生的概率，对于预防死锁的发生会有一定的作用。
- 降低隔离级别，如果业务允许，将隔离级别调低也是较好的选择
- 如果没有别的办法了，用表级锁定优化你的事务
- 死锁是无法完全避免的

Mysql中mvcc逻辑流程-插入

主键（自增） 姓名 年龄 数据行的版本号 删除版本号

表：teacher

id	name	age	DB_TRX_ID	DB_ROLL_PT

step：插入数据

```
假设系统的全局事务ID号从1开始；
begin;          -- 拿到系统的事务ID=1；
insert into teacher(name,age) VALUE ('seven',18);
insert into teacher(name,age) VALUE ('qingshan',19);
commit;
```

主键（自增） 姓名 年龄 数据行的版本号 删除版本号

表：teacher

id	name	age	DB_TRX_ID	DB_ROLL_PT
1	seven	18	1	NULL
2	qingshan	19	1	NULL

思考：

如果我直接执行

```
insert into teacher(name,age)
VALUE ('seven',18);

insert into teacher(name,age)
VALUE ('qingshan',19);
```

且 set autocommit = OFF;

Mysql中mvcc逻辑流程-删除

主键（自增） 姓名 年龄 数据行的版本号 删除版本号

表：teacher

id	name	age	DB_TRX_ID	DB_ROLL_PT
1	seven	18	1	NULL
2	qingshan	19	1	NULL

step：数据的删除

```
假设系统的全局事务ID号目前到了22

begin;          -- 拿到系统的事务ID=22；

delete teacher where id =2 ;

commit;
```

主键（自增） 姓名 年龄 数据行的版本号 删除版本号

表：teacher

id	name	age	DB_TRX_ID	DB_ROLL_PT
1	seven	18	1	NULL
2	qingshan	19	1	*22*

Mysql中mvcc逻辑流程-修改

主键（自增） 姓名 年龄 数据行的版本号 删除版本号

表：teacher

id	name	age	DB_TRX_ID	DB_ROLL_PT
1	seven	18	1	NULL
2	qingshan	19	1	22

step：修改操作

```
假设系统的全局事务ID号目前到了33
begin;          -- 拿到系统的事务ID=33；
update teacher set age = 19 where id =1 ;
commit;
```

修改操作是先做命中的数据行的copy，将原行数据的删除版本号的值设置为当前事务ID(33)

主键（自增） 姓名 年龄 数据行的版本号 删除版本号

表：teacher

id	name	age	DB_TRX_ID	DB_ROLL_PT
1	seven	18	1	33
2	qingshan	19	1	22
1	seven	19	33	NULL

copy

Mysql中mvcc逻辑流程-查询

表：teacher

id	name	age	DB_TRX_ID	DB_ROLL_PT
1	seven	18	1	33
2	qingshan	19	1	22
1	seven	19	33	NULL

step：查询操作

```
假设系统的全局事务ID号目前到了44
begin;          -- 拿到系统的事务ID=44；
select * from users ;
commit;
```

主键（自增）　姓名　年龄　数据行的版本号　删除版本号

表：teacher

id	name	age	DB_TRX_ID	DB_ROLL_PT
1	seven	18	1	33
2	qingshan	19	1	22
1	seven	19	33	NULL

1，seven，19

数据查询规则

1.查找数据行版本早于当前事务版本的数据行
　　(也就是,行的系统版本号小于或等于事务的系统版本号)，这样可以确保事务读取的行，要么是在事务开始前已经存在的，要么是事务自身插入或者修改过的

2.查找删除版本号要么为null，要么大于当前事务版本号的记录
　　确保取出来的行记录在事务开启之前没有被删除

Mysql中mvcc版本控制案例

事务1:

```
begin;
insert into mvcctest values(NULL,'Jerry') ;
insert into mvcctest values(NULL,'jack') ;
commit;
```

id	name	数据行的版本号	删除版本号
1	Jerry	1	Null
2	jack	1	Null

Mysql中mvcc版本控制案例

事务2:执行第 1 次查询，读取到两条原始数据，这个时候事务 ID 是 2：

```
begin;
select * from mvcctest ; -- (1) 第一次查询
```

id	name	数据行的版本号	删除版本号
1	Jerry	1	Null
2	jack	1	Null

Mysql中mvcc版本控制案例

事务3:插入数据：这个时候事务 ID 是 3

```
begin;
insert into mvcctest values(NULL,'tom') ;
commit;
```

id	name	数据行的版本号	删除版本号
1	Jerry	1	Null
2	jack	1	Null
3	tom	3	Null

事务2执行第二次查询：

select * from mvcctest ; (2) 第二次查询

id	name	数据行的版本号	删除版本号
1	Jerry	1	Null
2	jack	1	Null

Mysql中mvcc版本控制案例

事务4:删除数据：这个时候事务 ID 是 4

```
begin;
delete from mvcctest where id=2;
commit;
```

id	name	数据行的版本号	删除版本号
1	Jerry	1	Null
2	jack	1	4
3	tom	3	Null

事务2执行第三次查询：

select * from mvcctest ; (3) 第三次查询

id	name	数据行的版本号	删除版本号
1	Jerry	1	Null
2	jack	1	4

Mysql中mvcc版本控制案例

事务5:更新数据：这个时候事务 ID 是5

```
begin;
update mvcctest set name ='Mic' where id=1;
commit;
```

id	name	数据行的版本号	删除版本号
1	Jerry	1	5
2	jack	1	4
3	tom	3	Null
1	Mic	5	Null

事务2执行第四次查询：

select * from mvcctest ; (4) 第四次查询

id	name	数据行的版本号	删除版本号
1	Jerry	1	Null
2	jack	1	4

通过版本号的控制，无论其它事务是插入、修改、删除，第一个查询到的数据都没有变化。在Innodb中，mvcc和锁是协同使用的。

Undo Log

- Undo Log 是什么：
  - undo意为取消，以撤销操作为目的，返回指定某个状态的操作
  - undo log指事务开始之前，在操作任何数据之前,首先将需操作的数据备份到一个地方 (Undo Log)
- UndoLog是为了实现事务的原子性而出现的产物

- Undo Log实现事务原子性：
  - 事务处理过程中如果出现了错误或者用户执行了 ROLLBACK语句,Mysql可以利用Undo Log中的备份将数据恢复到事务开始之前的状态

- UndoLog在Mysql innodb存储引擎中用来实现多版本并发控制

- Undo log实现多版本并发控制：
  - 事务未提交之前，Undo保存了未提交之前的版本数据，Undo 中的数据可作为数据旧版本快照供其他并发事务进行快照读

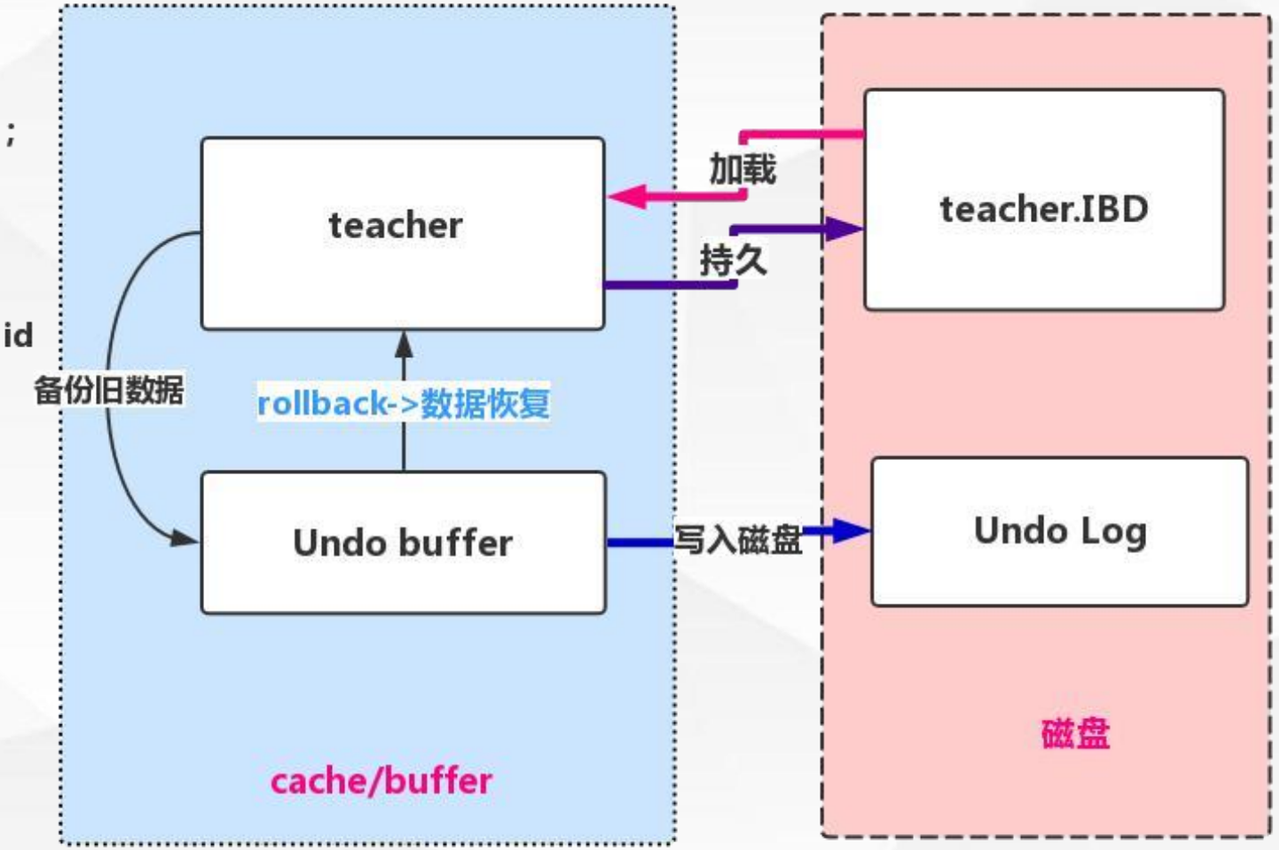
Undo Log

step1 事务A
begin ;
update teacher set age =20 ;
where id =1 ;

step2 事务B
select * from teacher where id =1 ;

//读取Undo中的数据进行返回，进行快照读

step3 事务A
commit/rollback



快照读、当前读

- 快照读：

  SQL读取的数据是快照版本，也就是历史版本，普通的SELECT就是快照读

  innodb快照读，数据的读取将由 cache(原本数据) + undo(事务修改过的数据) 两部分组成

- 当前读：

  SQL读取的数据是最新版本。通过锁机制来保证读取的数据无法通过其他事务进行修改 UPDATE、DELETE、INSERT、SELECT … LOCK IN SHARE MODE、SELECT … FOR UPDATE都是

Redo Log

- Redo Log 是什么：
  - Redo，顾名思义就是重做。以恢复操作为目的，重现操作；
  - Redo log指事务中操作的任何数据,将最新的数据备份到一个地方 (Redo Log)

- Redo log的持久：

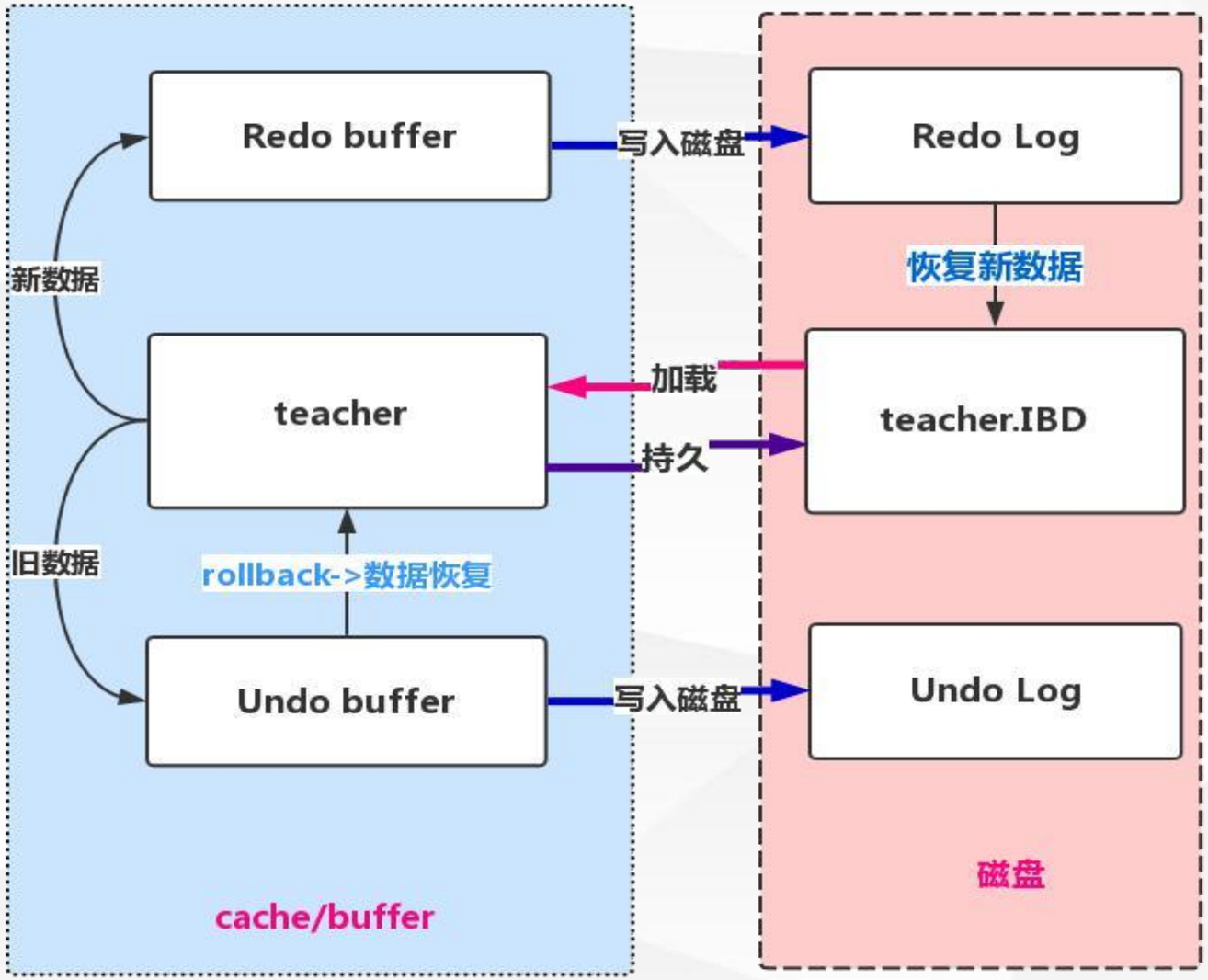
  不是随着事务的提交才写入的，而是在事务的执行过程中，便开始写入redo 中。具体 的落盘策略可以进行配置 redo log写入成功代表你sql执行成功。

- RedoLog是为了实现事务的持久性而出现的产物

- Redolog实现事务持久性：

- 防止在发生故障的时间点，断电，尚有脏页未写入磁盘，在重启mysql服务的时候，根据redo log进行重做，从而达到事务的未入磁盘数据进行持久化这一特性。

Redo Log


Redo buffer
写入磁盘
Redo Log
新数据
恢复新数据
teacher
加载
持久
teacher.IBD
旧数据
rollback->数据恢复
Undo buffer
写入磁盘
Undo Log
cache/buffer
磁盘

Redo Log & bin Log

- redo log是属于innoDB层面，binlog属于MySQL Server层面的，这样在数据库用别的存储引擎时可以达到一致性的要求。
- redo log是物理日志，记录该数据页更新的内容；binlog是逻辑日志，记录的是这个更新语句的原始逻辑
- redo log是循环写，日志空间大小固定；binlog是追加写，是指一份写到一定大小的时候会更换下一个文件，不会覆盖。
- binlog可以作为恢复数据使用，主从复制搭建，redo log作为异常宕机或者介质故障后的数据恢复使用。

第5部分

主从复制、分库分表