InnoDB Architecture

李力

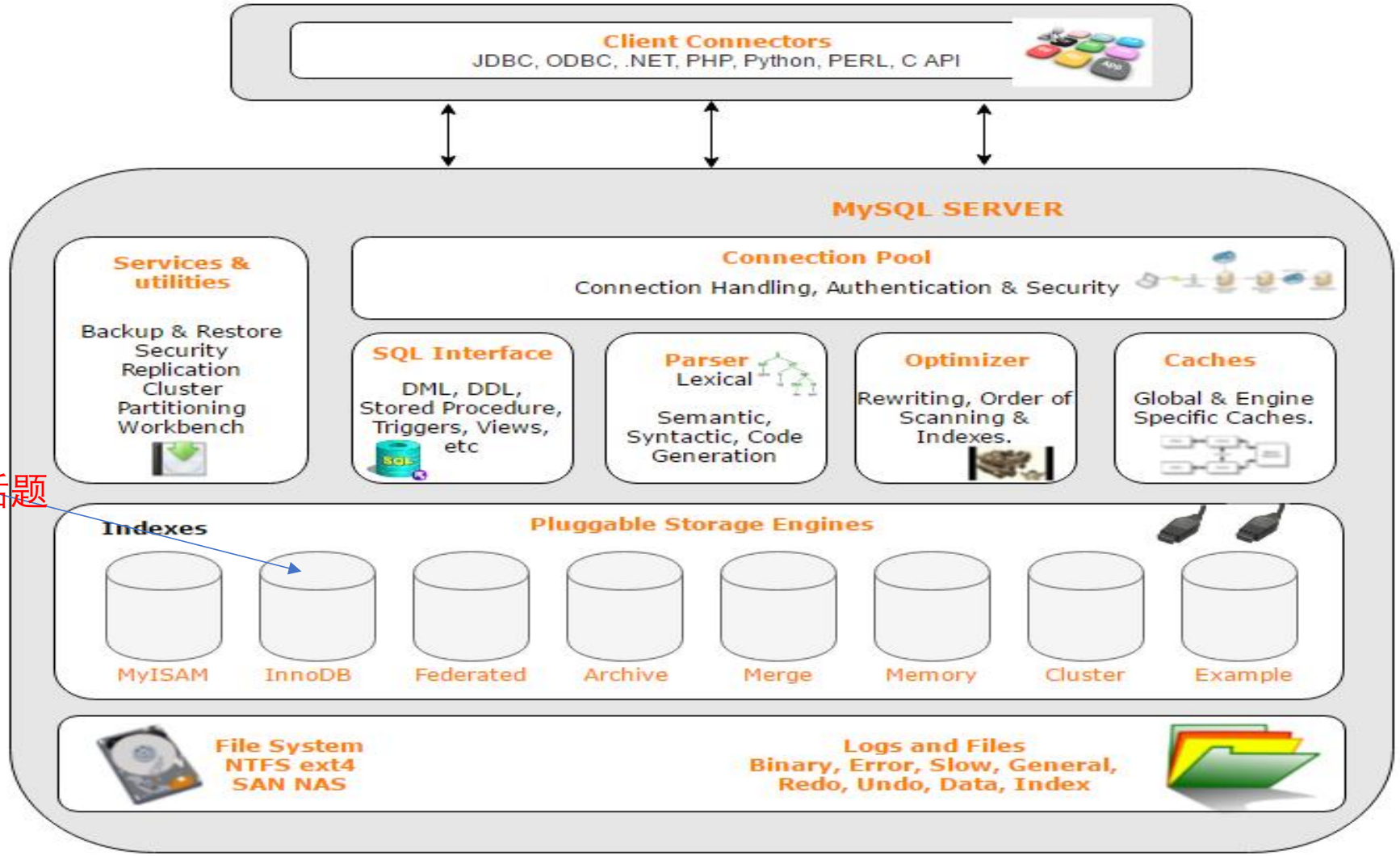
简单的自测

- 数据库中的锁有哪些?
- 事务隔离级别有哪些？不同级别为什么会导致数据不一致的?
- 事务到底怎么实现的?
- ACID和JMM联系?
- InnoDB内存结构怎么样?
- 常见索引种类?
- 宕机怎么保证数据不丢?
- 如何解决幻读问题?
- Redis，MongoDB，HBase?
- InndoDB究竟能存多大数据？最多建多少张表，一张表最多多少字段?

大纲

- 1 MySQL Architecture
- 2 InnoDB Architecture
  - 2.1 结构体系和特点
  - 2.2 说过程
  - 2.3 说存储
  - 2.4 说过程
- 3 Index
- 4 Transcation
- 5 Optimization

MySQL Architecture

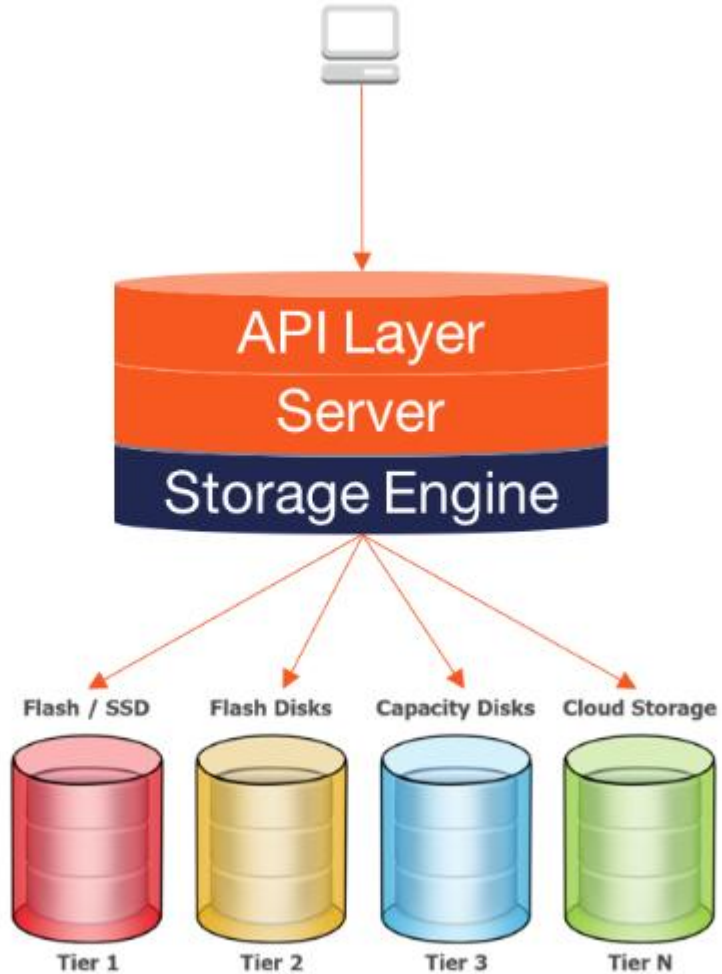


今天讨论的话题

App Client

Database

Memory
&
Storage

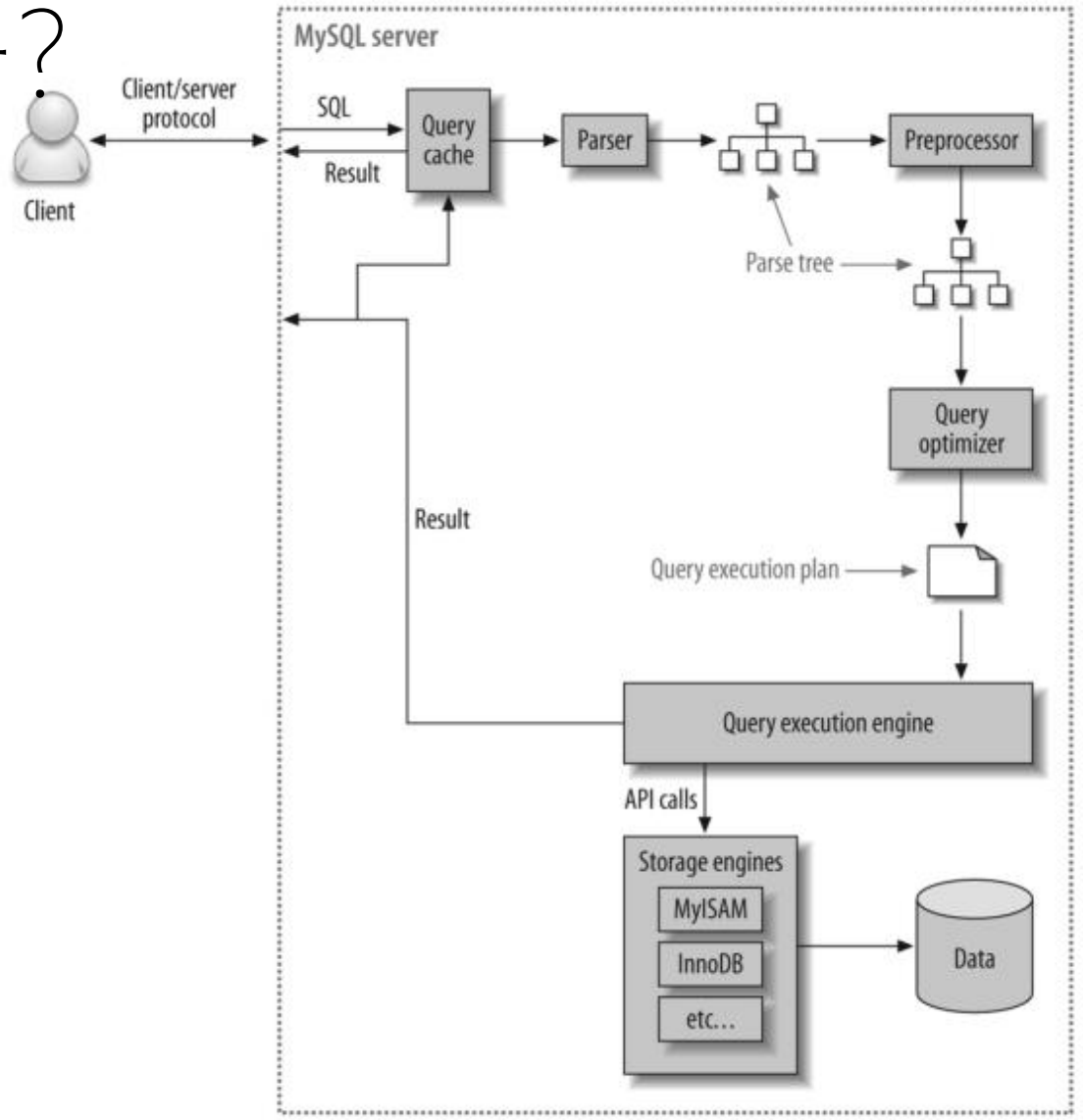


体系结构是理解MySQL的起点

- 1 插件存储体系
- 2 单进程多线程的CS结构
- 3 分层体系(连接层，服务器层，存储引擎层)
  - 3.1 连接层:连接处理(AIO)，安全，授权
  - 3.2 服务器层：查询解析，分析，优化，内置函数，存储过程，触发器，视图
  - 3.3 存储引擎层：事务，MVCC，内存，磁盘管理，索引实现

  - 思考：SQL优化在哪一层呢？Lock呢？排序呢？

SQL如何被执行？

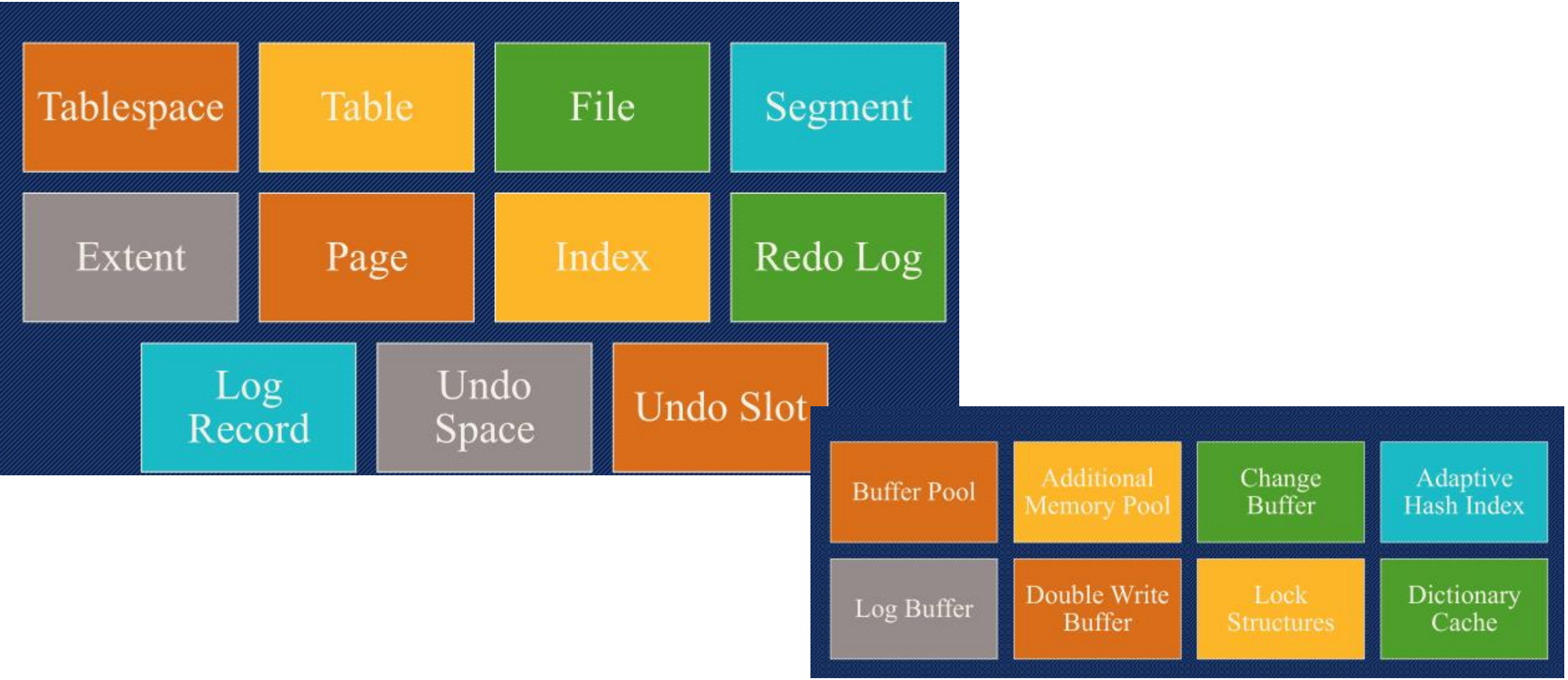

MySQL server
Client/server protocol
Client
SQL
Result
Query cache
Parser
Preprocessor
Parse tree
Query optimizer
Result
Query execution plan
Query execution engine
API calls
Storage engines
MyISAM
InnoDB
etc...
Data

```
mysql> show engines;
+--------------------+---------+----------------------------------------------------------------+--------------+------+------------+
| Engine             | Support | Comment                                                        | Transactions | XA   | Savepoints |
+--------------------+---------+----------------------------------------------------------------+--------------+------+------------+
| MEMORY             | YES     | Hash based, stored in memory, useful for temporary tables      | NO           | NO   | NO         |
| MRG_MYISAM         | YES     | Collection of identical MyISAM tables                          | NO           | NO   | NO         |
| CSV                | YES     | CSV storage engine                                             | NO           | NO   | NO         |
| FEDERATED          | NO      | Federated MySQL storage engine                                 | NULL         | NULL | NULL       |
| PERFORMANCE_SCHEMA | YES     | Performance Schema                                             | NO           | NO   | NO         |
| MyISAM             | YES     | MyISAM storage engine                                          | NO           | NO   | NO         |
| InnoDB             | DEFAULT | Supports transactions, row-level locking, and foreign keys     | YES          | YES  | YES        |
| BLACKHOLE          | YES     | /dev/null storage engine (anything you write to it disappears) | NO           | NO   | NO         |
| ARCHIVE            | YES     | Archive storage engine                                         | NO           | NO   | NO         |
+--------------------+---------+----------------------------------------------------------------+--------------+------+------------+
```

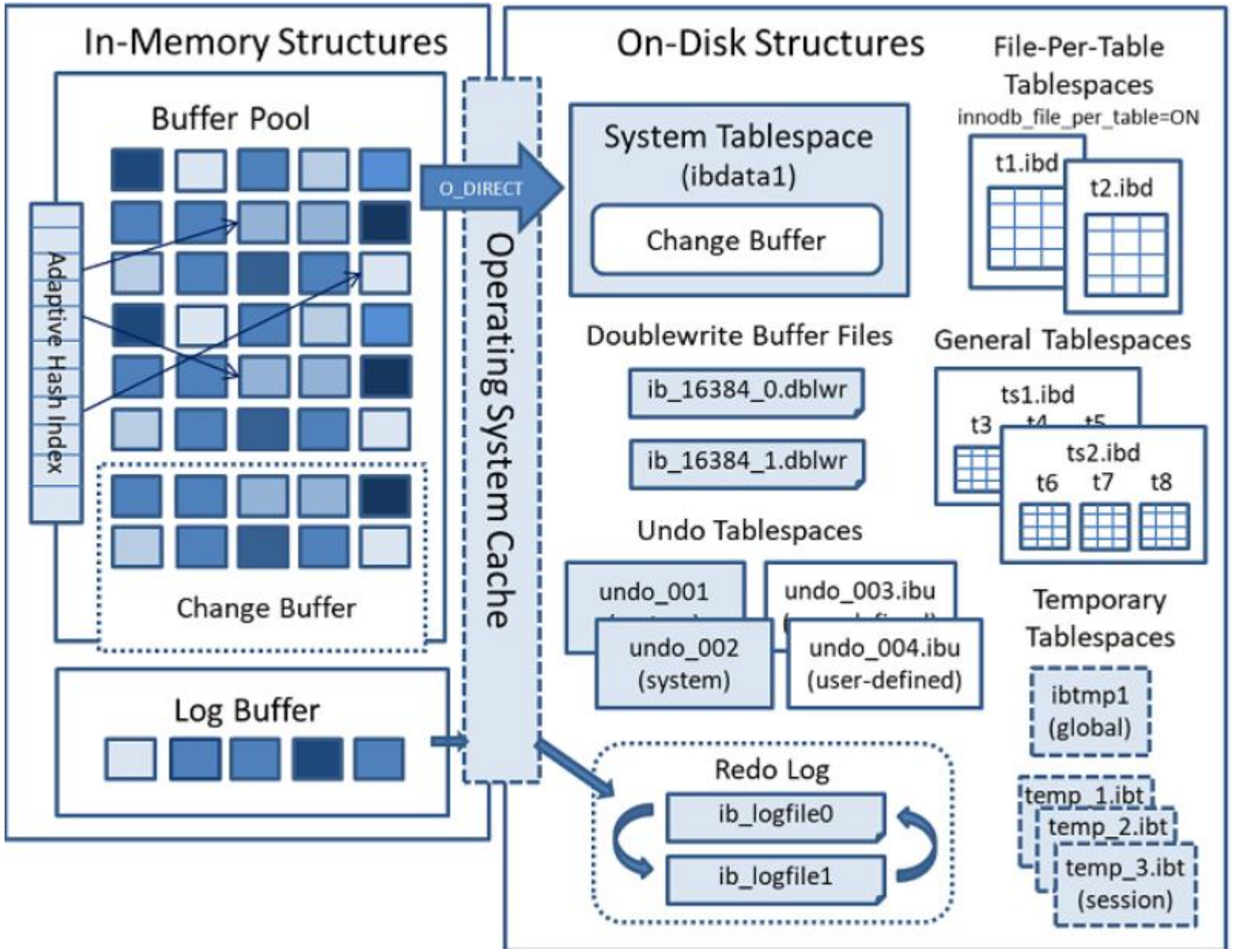
2.1说特性

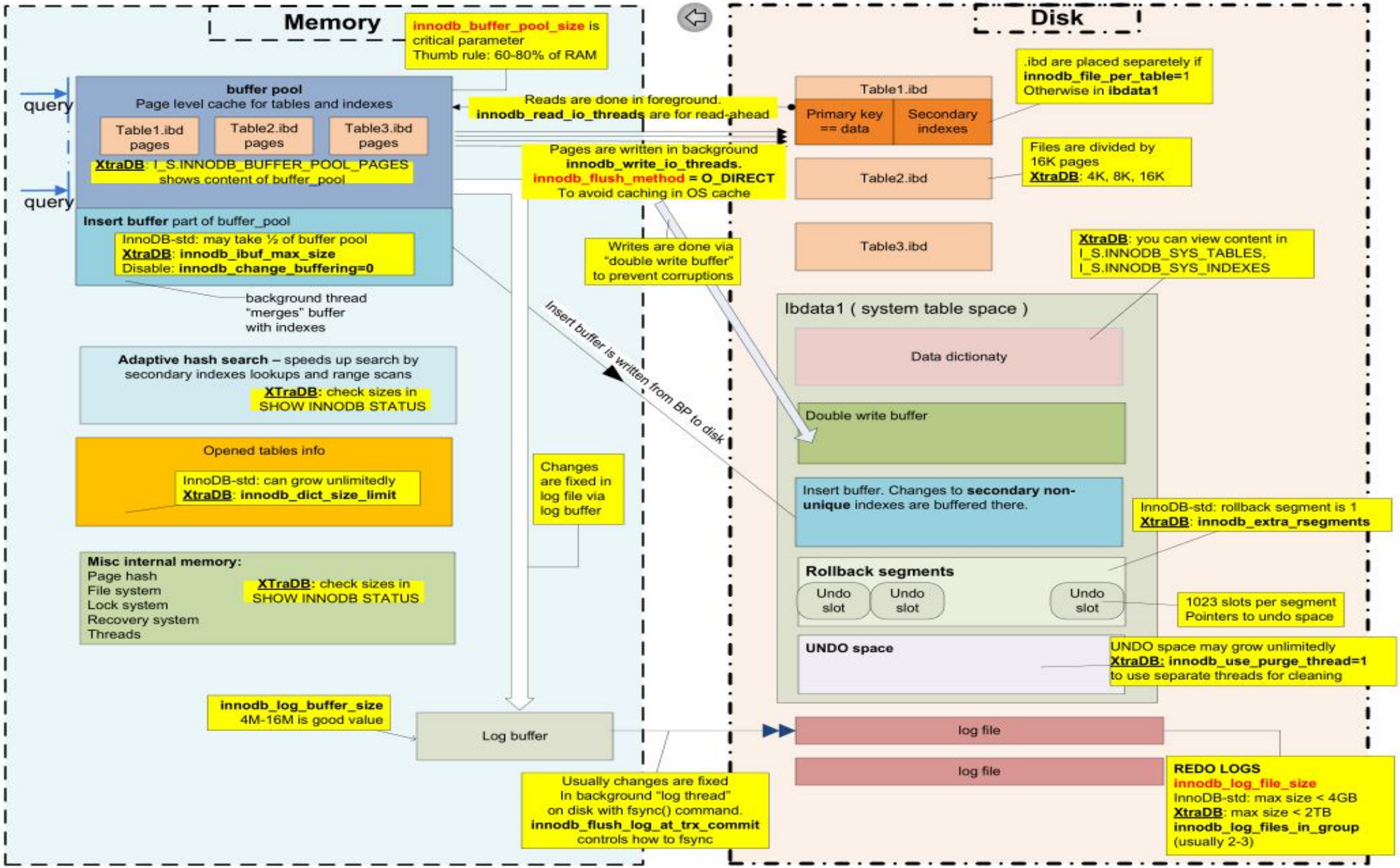
- 1 支持事务，ACID
- 2 行锁，外键，一致性非锁定读，MVCC
- 3 change buffer, double write, adaptive hash, read ahead
- 4 支持聚簇索引和非聚簇索引
- 5 高效利用内存和CPU

InnoDB “Object”



InnoDB Architecture




Memory
innodb_buffer_pool_size is critical parameter
Thumb rule: 60-80% of RAM
query
query
buffer pool
Page level cache for tables and indexes
Table1.ibd pages
Table2.ibd pages
Table3.ibd pages
XtraDB: I_S.INNODB_BUFFER_POOL_PAGES shows content of buffer_pool
Insert buffer part of buffer_pool
InnoDB-std: may take ½ of buffer pool
XtraDB: innodb_ibuf_max_size
Disable: innodb_change_buffering=0
background thread "merges" buffer with indexes
Adaptive hash search – speeds up search by secondary indexes lookups and range scans
XTraDB: check sizes in SHOW INNODB STATUS
Opened tables info
InnoDB-std: can grow unlimitedly
XtraDB: innodb_dict_size_limit
Misc internal memory:
Page hash
File system
Lock system
Recovery system
Threads
XTraDB: check sizes in SHOW INNODB STATUS
innodb_log_buffer_size
4M-16M is good value
Log buffer
Reads are done in foreground.
innodb_read_io_threads are for read-ahead
Pages are written in background
innodb_write_io_threads.
innodb_flush_method = O_DIRECT
To avoid caching in OS cache
Writes are done via "double write buffer" to prevent corruptions
Insert buffer is written from BP to disk
Changes are fixed in log file via log buffer
Usually changes are fixed In background "log thread" on disk with fsync() command.
innodb_flush_log_at_trx_commit controls how to fsync
Disk
Table1.ibd
Primary key == data
Secondary indexes
.ibd are placed separetely if innodb_file_per_table=1
Otherwise in ibdata1
Table2.ibd
Files are divided by 16K pages
XtraDB: 4K, 8K, 16K
Table3.ibd
XtraDB: you can view content in I_S.INNODB_SYS_TABLES, I_S.INNODB_SYS_INDEXES
Ibdata1 (system table space)
Data dictionaty
Double write buffer
Insert buffer. Changes to secondary non-unique indexes are buffered there.
InnoDB-std: rollback segment is 1
XtraDB: innodb_extra_rsegments
Rollback segments
Undo slot
Undo slot
Undo slot
1023 slots per segment
Pointers to undo space
UNDO space
UNDO space may grow unlimitedly
XtraDB: innodb_use_purge_thread=1
to use separate threads for cleaning
log file
log file
REDO LOGS
innodb_log_file_size
InnoDB-std: max size < 4GB
XtraDB: max size < 2TB
innodb_log_files_in_group
(usually 2-3)

2.1 说过程之后台线程

- 线程
  - Master Thread
  - IO Thread（4R4W）
  - Purge Thread
  - Page Clean Thread

  mysql>show engine innodb status\G;
  查看MySQL负载

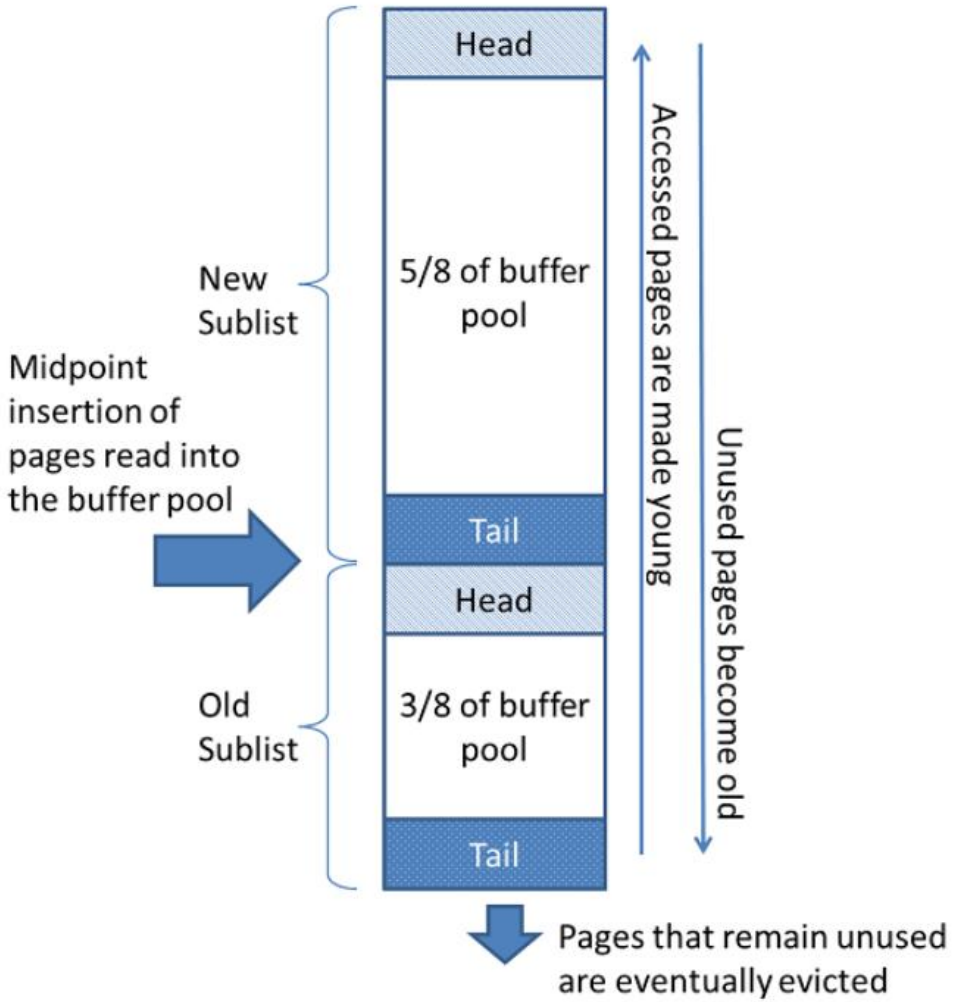
```
BACKGROUND THREAD
-----------------
srv_master_thread loops: 84 srv_active, 0 srv_shutdown, 500876 srv_idl
srv_master_thread log flush and writes: 0
----------
```

```
--------
FILE I/O
--------
I/O thread 0 state: wait Windows aio (insert buffe
I/O thread 1 state: wait Windows aio (log thread)
I/O thread 2 state: wait Windows aio (read thread)
I/O thread 3 state: wait Windows aio (read thread)
I/O thread 4 state: wait Windows aio (read thread)
I/O thread 5 state: wait Windows aio (read thread)
I/O thread 6 state: wait Windows aio (write thread
I/O thread 7 state: wait Windows aio (write thread
I/O thread 8 state: wait Windows aio (write thread
I/O thread 9 state: wait Windows aio (write thread
Pending normal aio reads: [0, 0, 0, 0] , aio write
 ibuf aio reads:, log i/o's:, sync i/o's:
Pending flushes (fsync) log: 0; buffer pool: 0
1112 OS file reads, 37288 OS file writes, 3406 OS
0.00 reads/s, 0 avg bytes/read, 0.00 writes/s, 0.0
-------------------------------------
```

```
void master_thread(){
   goto loop;
loop:
for(int i = 0; i<10; i++){
   thread_sleep(1) // sleep 1 second
   do log buffer flush to disk
   if ( last_one_second_ios < 5% innodb_io_capacity )
      do merge 5% innodb_io_capacity insert buffer
   if ( buf_get_modified_ratio_pct > innodb_max_dirty_pages_pct )
      do buffer pool flush 100% innodb_io_capacity dirty page
   else if enable adaptive flush
      do buffer pool flush desired amount dirty page
   if ( no user activity )
      goto backgroud loop
}
if ( last_ten_second_ios <innodb_io_capacity)
   do buffer pool flush 100% innodb_io_capacity dirty page
do merge 5% innodb_io_capacity insert buffer
do log buffer flush to disk
do full purge
if ( buf_get_modified_ratio_pct > 70% )
   do buffer pool flush 100% innodb_io_capacity dirty page
else
   dobuffer pool flush 10% innodb_io_capacity dirty page
goto loop
background loop:
do full purge
do merge 100% innodb_io_capacity insert buffer
if not idle:
goto loop:
else:
   goto flush loop
flush loop:
do buffer pool flush 100% innodb_io_capacity dirty page
if ( buf_get_modified_ratio_pct>innodb_max_dirty_pages_pct )
   go to flush loop
   goto suspend loop
suspend loop:
suspend_thread()
waiting event
goto loop;
}
```

2.2 说存储之内存结构

- 内存之**Buffer Pool List**
- **淘汰之后去被刷新到磁盘了**



2.2 说存储之内存结构

```
----------------------
BUFFER POOL AND MEMORY
----------------------
Total large memory allocated 137363456
Dictionary memory allocated 972926
Buffer pool size   8192
Free buffers       6393
Database pages     1720
Old database pages 614
Modified db pages  0
Pending reads      0
Pending writes: LRU 0, flush list 0, single page 0
Pages made young 445, not young 0
0.00 youngs/s, 0.00 non-youngs/s
Pages read 1087, created 642, written 2118
0.00 reads/s, 0.00 creates/s, 0.00 writes/s
No buffer pool page gets since the last printout
Pages read ahead 0.00/s, evicted without access 0.00/s, Random read ahead 0.00/s
LRU len: 1720, unzip_LRU len: 0
I/O sum[0]:cur[0], unzip sum[0]:cur[0]
--------------
```

2.2 说存储之内存结构

- 内存之**Insert Buffer -> Change Buffer**
  - **解决非聚集叶子节点插入无序问题,不直接插入索引页，而是先判断 Buffer Pool**
  - **满足条件：1 索引是secondary index 2 索引不是unique**
  - Change Buffer: INSERT（Insert Buffer）,DELETE(Delete Buffer),UPDATE(Purge Buffer)

2.2 说存储之内存结构

- 内存之Adaptive Hash Index

- 联合索引(a,b)
- Where a=xxx;
- Where a=xxx and b =xxx
- 以该模式访问100次
- 交替执行不会构建AHI
- 页通过该模式访问N次，N=页中记录*1/16

```
-------------------------------------
INSERT BUFFER AND ADAPTIVE HASH INDEX
-------------------------------------
Ibuf: size 1, free list len 0, seg size 2, 0 merges
merged operations:
 insert 0, delete mark 0, delete 0
discarded operations:
 insert 0, delete mark 0, delete 0
Hash table size 34679, node heap has 4 buffer(s)
Hash table size 34679, node heap has 2 buffer(s)
Hash table size 34679, node heap has 25 buffer(s)
Hash table size 34679, node heap has 1 buffer(s)
Hash table size 34679, node heap has 2 buffer(s)
Hash table size 34679, node heap has 8 buffer(s)
Hash table size 34679, node heap has 4 buffer(s)
Hash table size 34679, node heap has 33 buffer(s)
0.00 hash searches/s, 0.00 non-hash searches/s
```

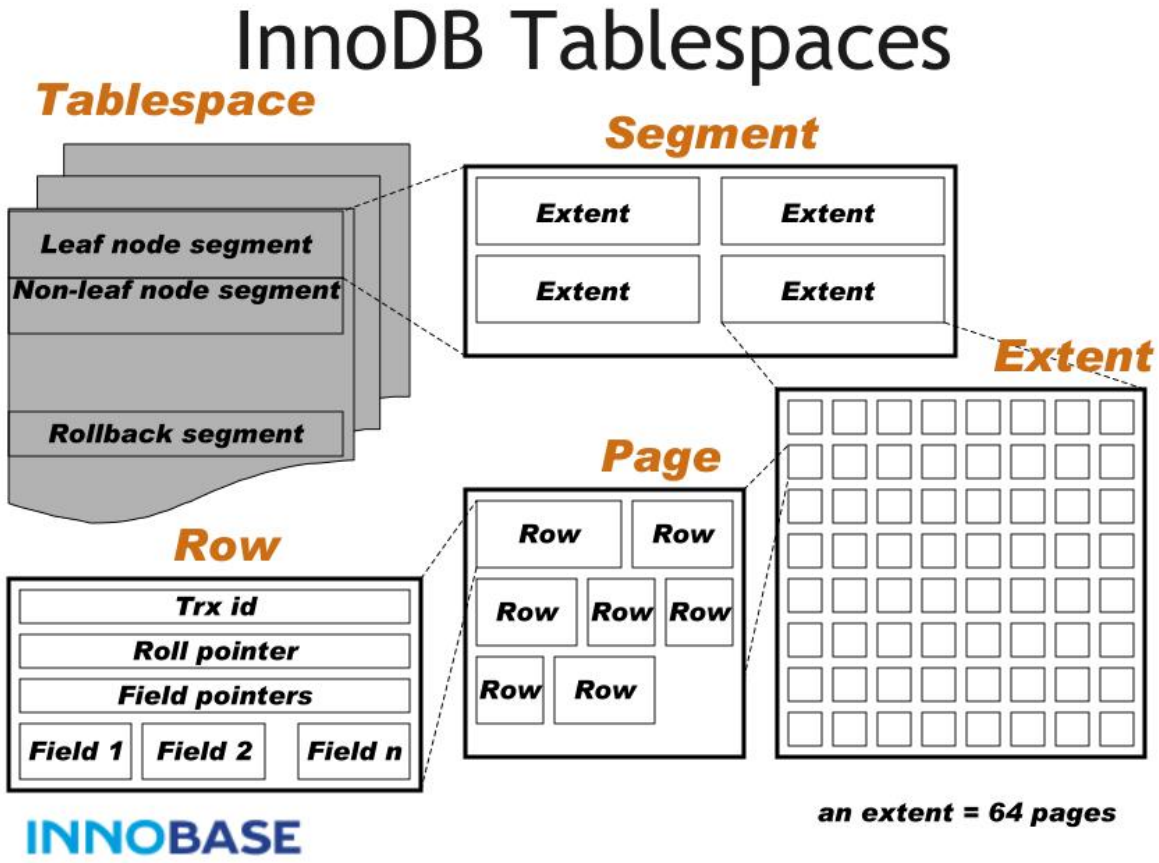
2.2 说存储之内存结构

- 内存之**Log Buffer**
- **Redo log:WAL+刷新脏页**
- **ACID**
- **innodb_flush_log_at_trx_commit**
  - **0 write and flush once per second**
  - **1 write and flush at each commit**
  - **2 write at commit，flush one per second**

```
---
LOG
---
Log sequence number          290684217
Log buffer assigned up to    290684217
Log buffer completed up to   290684217
Log written up to            290684217
Log flushed up to            290684217
Added dirty pages up to      290684217
Pages flushed up to          290684217
Last checkpoint at           290684217
34634 log i/o's done, 0.00 log i/o's/second
```

2.2 说存储之磁盘

- 磁盘之tablespaces

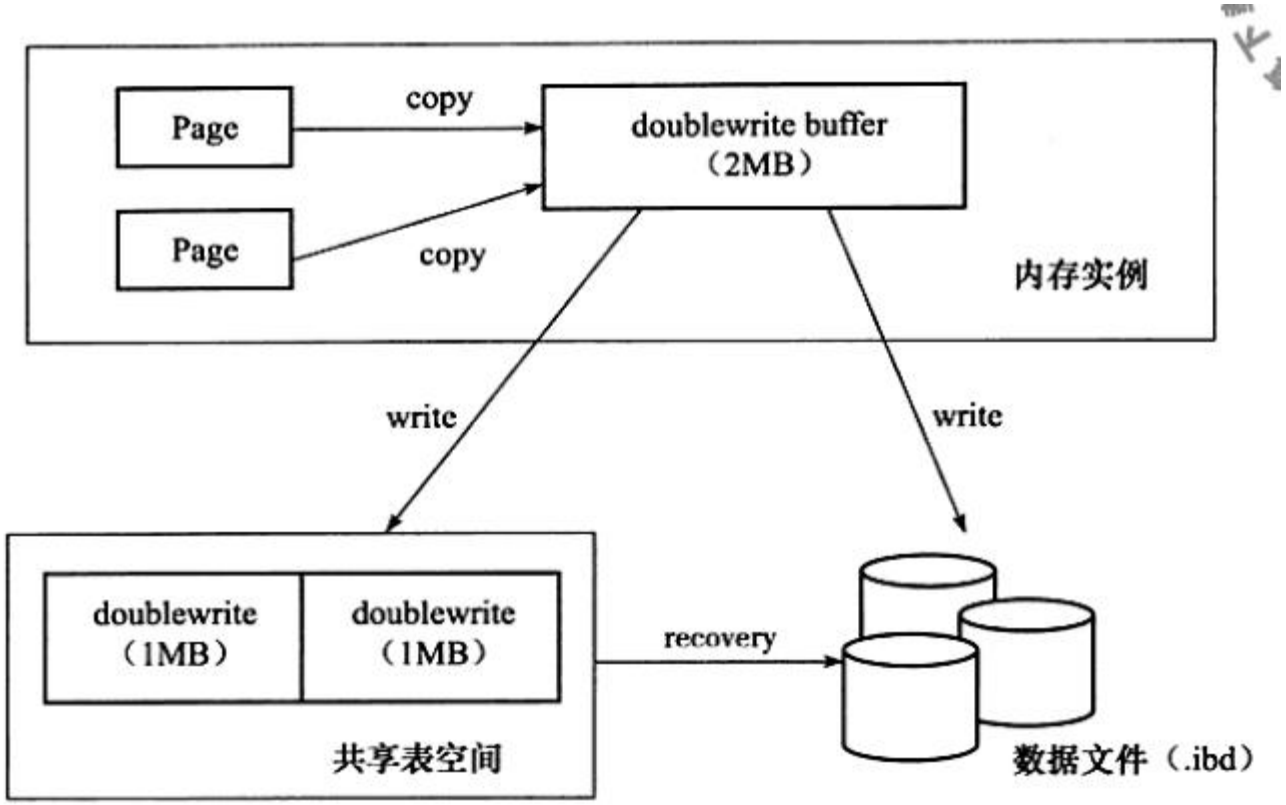


2.2 说存储之磁盘

- 磁盘之Doublewrite Buffer
  - 保证数据页的可靠性
  - 16K页写了4K，宕机，

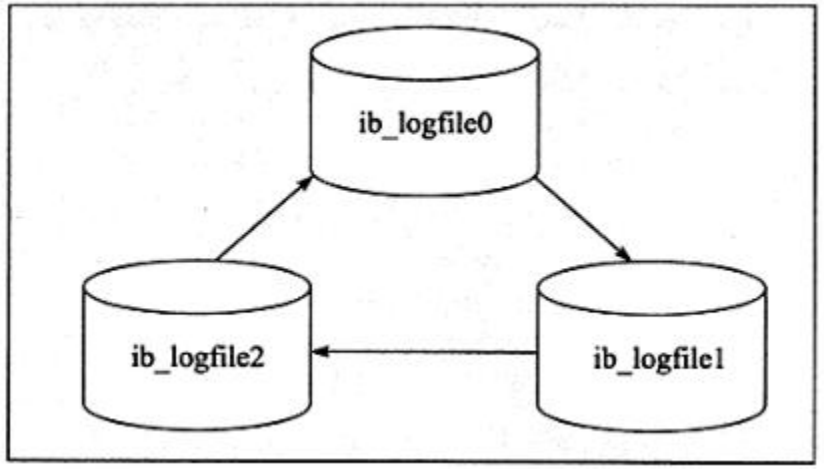
    Redo操作基于页，页损坏

    Redo没用，写入共享表空间，宕机重启复制到表空间，然后redo



2.2 说存储之磁盘

- 磁盘之Redo Log
  - 保证数据完整性
  - 实现事务持久性



2.2 说存储之磁盘

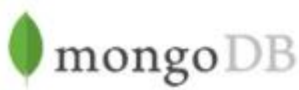
- 磁盘之Undo Log

3 Index

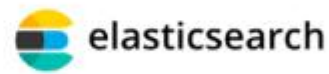
- 结构分类： hash，b+ tree，r tree，fractal tree index(Toku DB), lsm tree(rocksdb,leveldb), Inverted index（elasticsearch，solr）
- Why B+ tree？ 优点： 1 减少服务器扫描数据量 2 避免服务器排序和临时表 3 随机IO变成顺序IO
- 缺点：1 额外占磁盘空间 2 增加使用复杂度
- B+ tree索引分类：1 主键索引(聚簇索引) 2 唯一索引(聚簇索引) 3 普通索引(非聚簇索引-大部分索引)

DATABASE STORAGE ENGINES

B-TREE



LSM TREE



Saas线上库索引大小(2019-12-16统计)

```
1 •   SELECT   table_schema,
2              SUM(data_length+index_length)/1024/1024/1024 AS total_GB,
3              SUM(data_length)/1024/1024/1024 AS data_GB,
4              SUM(index_length)/1024/1024/1024 AS index_GB,
5              COUNT(*) AS tables,
6              CURDATE() AS today
7     FROM     information_schema.tables
8     GROUP BY table_schema
9     ORDER BY 2 DESC;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

table_schema	total_GB	data_GB	index_GB	tables	today
exchange_arch	4650.386886596680	2509.172851562500	2141.214035034180	48	2019-12-16
exchange	918.626190185547	467.741119384766	450.885070800781	4451	2019-12-16
contract	47.849288940430	23.676971435547	24.172317504883	96	2019-12-16
exchange_kline	42.083770751953	33.575637817383	8.508132934570	5618	2019-12-16
exchange_account	25.523269653320	19.220550537109	6.302719116211	14	2019-12-16
exchange_risk	13.338562011719	10.685913085938	2.652648925781	4254	2019-12-16
exchange_otc	7.091262817383	6.915664672852	0.175598144531	22	2019-12-16
contract_cryptoflex	0.944335937500	0.595870971680	0.348464965820	39	2019-12-16
contract_kline	0.687088012695	0.564041137695	0.123046875000	56	2019-12-16
security	0.534576416016	0.312530517578	0.222045898438	47	2019-12-16
schedule	0.435302734375	0.421600341797	0.013702392578	17	2019-12-16
contract_coinbus	0.217208862305	0.124160766602	0.093048095703	40	2019-12-16
hayek	0.194183349609	0.096923828125	0.097259521484	8	2019-12-16
contract_fubi	0.150802612305	0.083786010742	0.067016601563	44	2019-12-16
contract_coinbus_...	0.018997192383	0.015991210938	0.003005981445	12	2019-12-16
qt20190713	0.011444091797	0.011398315430	0.000045776367	4	2019-12-16

Saas线上表索引大小(2019-12-16统计)

```sql
SELECT table_schema,table_name, table_rows,
       ROUND((data_length+index_length)/1024/1024/1024) AS total_GB,
       ROUND(data_length/1024/1024/1024) AS data_GB,
       ROUND(index_length/1024/1024/1024) AS index_GB
FROM INFORMATION_SCHEMA.TABLES
WHERE engine='InnoDB' order by 4 desc;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content: | Fetch rows:

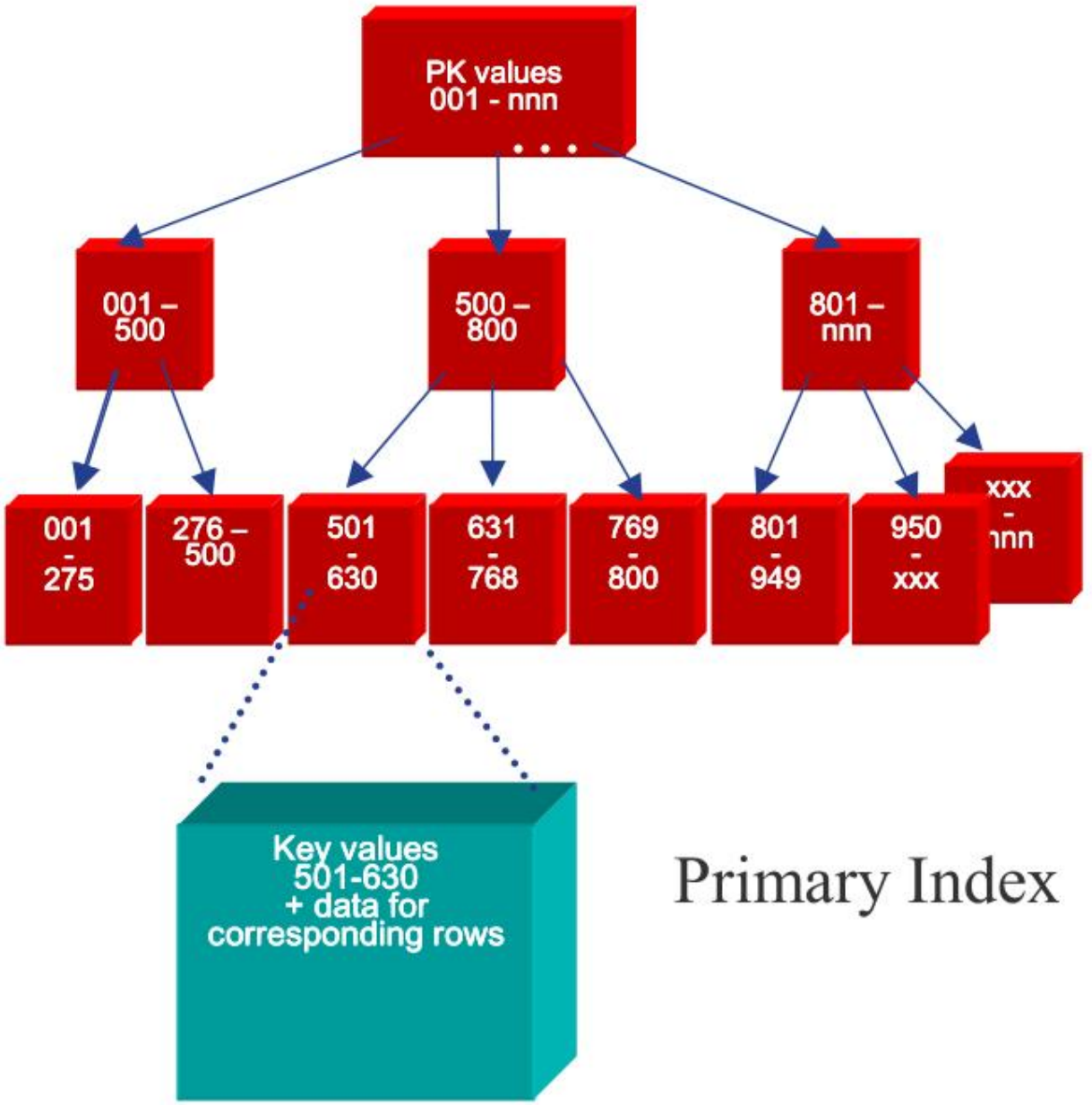
table_schema	table_name	table_rows	total_GB	data_GB	index_GB
exchange_arch	transaction_2019111210	1375485785	649	376	273
exchange_arch	transaction_2019080420	1099484681	424	227	196
exchange_arch	transaction_2019092114	743663527	291	157	134
exchange_arch	transaction_2019053010	627525139	237	127	109
exchange	transaction	424786469	213	122	91
exchange_arch	transaction_old3	543528719	205	111	94
exchange_arch	transaction_old9	460815976	175	94	80
exchange_arch	transaction_old2	449553405	171	90	81
exchange_arch	transaction_old1	546431624	164	92	72
exchange_arch	transaction_old6	404456761	155	83	72
exchange_arch	transaction_old7	380090567	147	79	68
exchange_arch	ex_order_client_20191211	543482579	140	69	71
exchange_arch	transaction_old4	341209173	131	70	61
exchange_arch	transaction_2019051010	301831829	116	62	54
exchange_arch	transaction_old8	281915469	109	58	50
exchange_arch	transaction_2019081710	262131239	103	55	48
exchange_arch	transaction_old_201904...	252315281	97	52	45
exchange	ex_order_client	411955192	88	43	45
exchange_arch	transaction_old5	220568713	85	45	39
exchange_arch	transaction_old10	166304973	63	34	30
exchange_arch	transaction_2019050210	115636299	45	24	21
exchange_arch	transaction_old13	115532086	45	24	21

3 Index

- Primary – Cluster Index

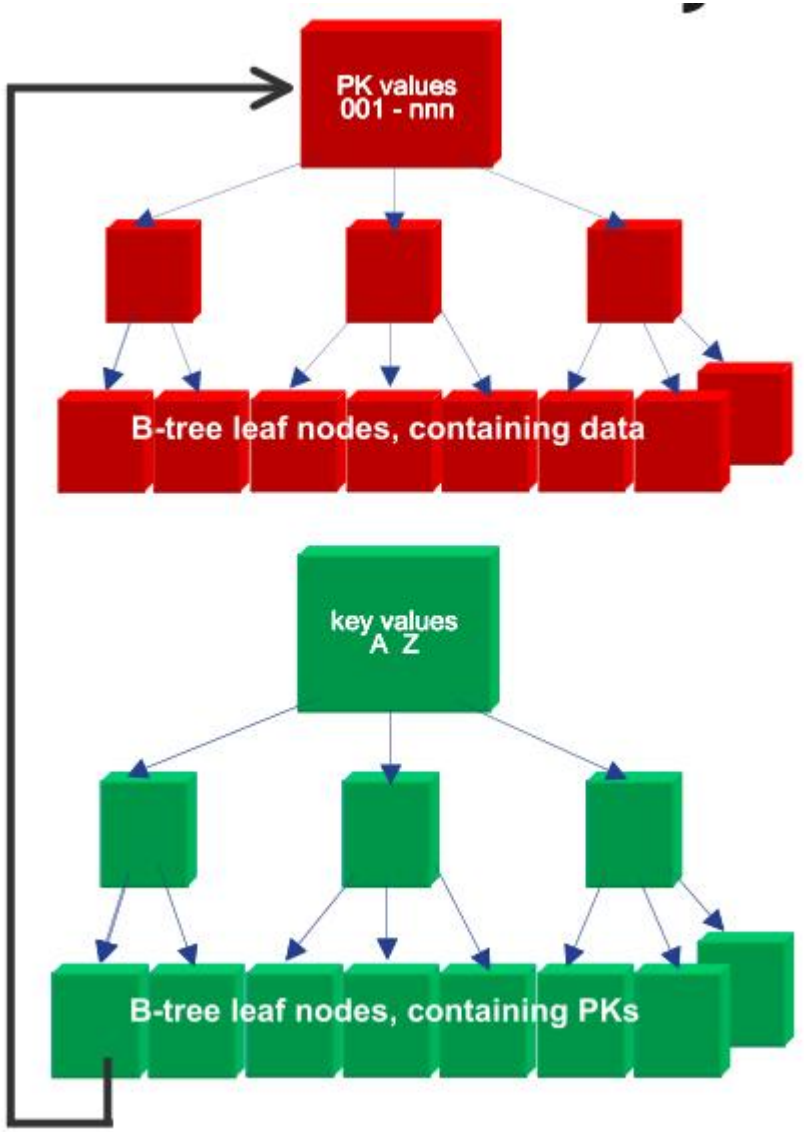
- PRIMARY KEY or
- UNIQUE INDEX or
- 6-byte ROW_ID

- 一张表只能有一个



3 Index

- Secondary – NoCluster Index
- INDEX
- 一张表0个，1个或多个



```
CREATE TABLE `exchange`.`wallet_withdraw_record` (
    PRIMARY KEY (`id`),
    UNIQUE KEY `uk_trans_id` (`trans_id`),
    UNIQUE KEY `uk_wallet_id` (`wallet_id`),
    KEY `idx_status` (`status`)
)  ENGINE=INNODB AUTO_INCREMENT=1 COMMENT='钱包提现流水记录';
```

- Select * from wallet_withdraw_record where id=1 树结构?
- Select * from wallet_withdraw_record where trans_id=1 树结构?
- Select trans_id from wallet_withdraw_record where trans_id=1 树结构?
- Select * from wallet_withdraw_record where idx_status=1 树结构?

clustered index

The InnoDB term for a ***primary key*** index. InnoDB table storage is organized based on the values of the primary key columns, to speed up queries and sorts involving the primary key columns. For best performance, choose the primary key columns carefully based on the most performance-critical queries. Because modifying the columns of the clustered index is an expensive operation, choose primary columns that are rarely or never updated.

secondary index

A type of InnoDB ***index*** that represents a subset of table columns. An InnoDB table can have zero, one, or many secondary indexes. (Contrast with the ***clustered index***, which is required for each InnoDB table, and stores the data for all the table columns.)

covering index

An ***index*** that includes all the columns retrieved by a query. Instead of using the index values as pointers to find the full table rows, the query returns values from the index structure, saving disk I/O. InnoDB can apply this optimization technique to more indexes than MyISAM can, because InnoDB ***secondary indexes*** also include the ***primary key*** columns. InnoDB cannot apply this technique for queries against tables modified by a transaction, until that transaction ends.

Any ***column index*** or ***composite index*** could act as a covering index, given the right query. Design your indexes and queries to take advantage of this optimization technique wherever possible.

4 Transaction

- 事务是什么？
  - 如何以最快速度完成事务是人类对数据库最大的追求。

- 事务特性：ACID 和 BASE
  - A：事务操作要么全做，要么全不做。
  - C：事务开始前处于一致性状态，结束后，处于一致性状态。
  - I：事务不受其他并发执行事务影响。
  - D：事务一旦完成，对数据库修改永久，系统故障也不会丢失。

- 事务和锁

事务单元

- 事务是让很多步操作顺序发生，多线程看起来是一步操作。-**并发调度的可串行性**
- 四种Happen-Before关系：RR，RW，WR，WW

```
public class LockTest {
    private String share = "我是大家共享的内存变量";
    private ReentrantLock lock = new ReentrantLock();
    public String doSomething(String args) {
        lock.lock();
        try {
            String tempStr = share + args;
            this.share = tempStr;
            return tempStr;
        } finally {
            lock.unlock();
        }
    }
}
```

处理事务方法

- 排队法-单线程
- 排它锁
- 读写锁
- MVCC

事务-排队法


事务—排队法
• 排队
• 序列化读写
• 优势
• 不需要冲突控制
• 劣势
• 慢速设备。。。
只读事务
读写事务
读写事务
只读事务
事务单元1
事务单元2
(2,3单元共享数据)
事务单元3

事务-排他锁

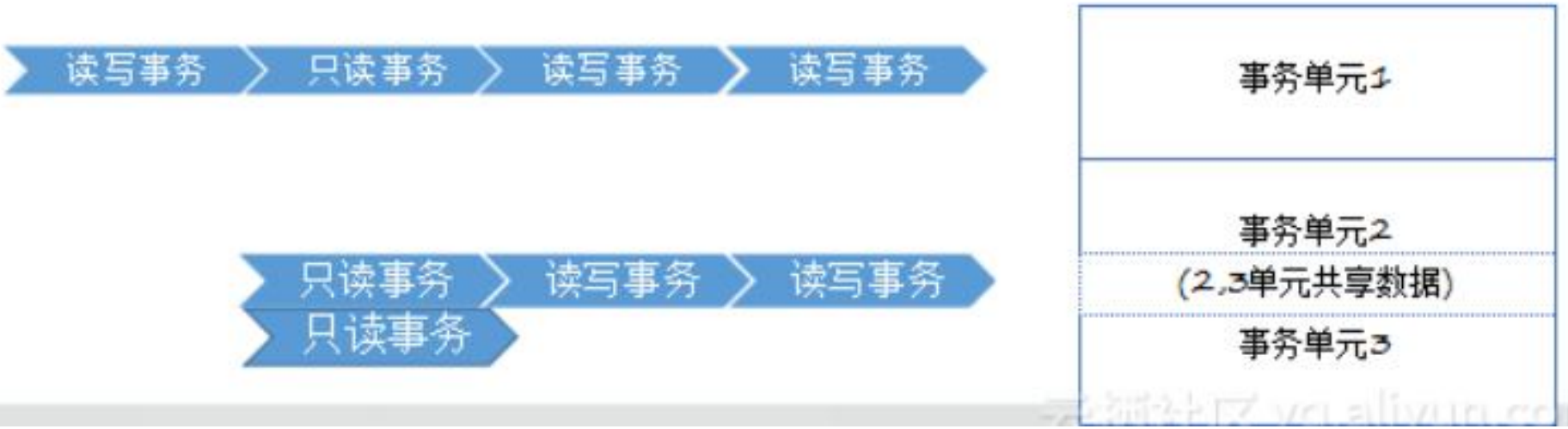
事务—排他锁

- 针对同一个单元的访问进行访问控制

事务-读写锁

事务—读写锁

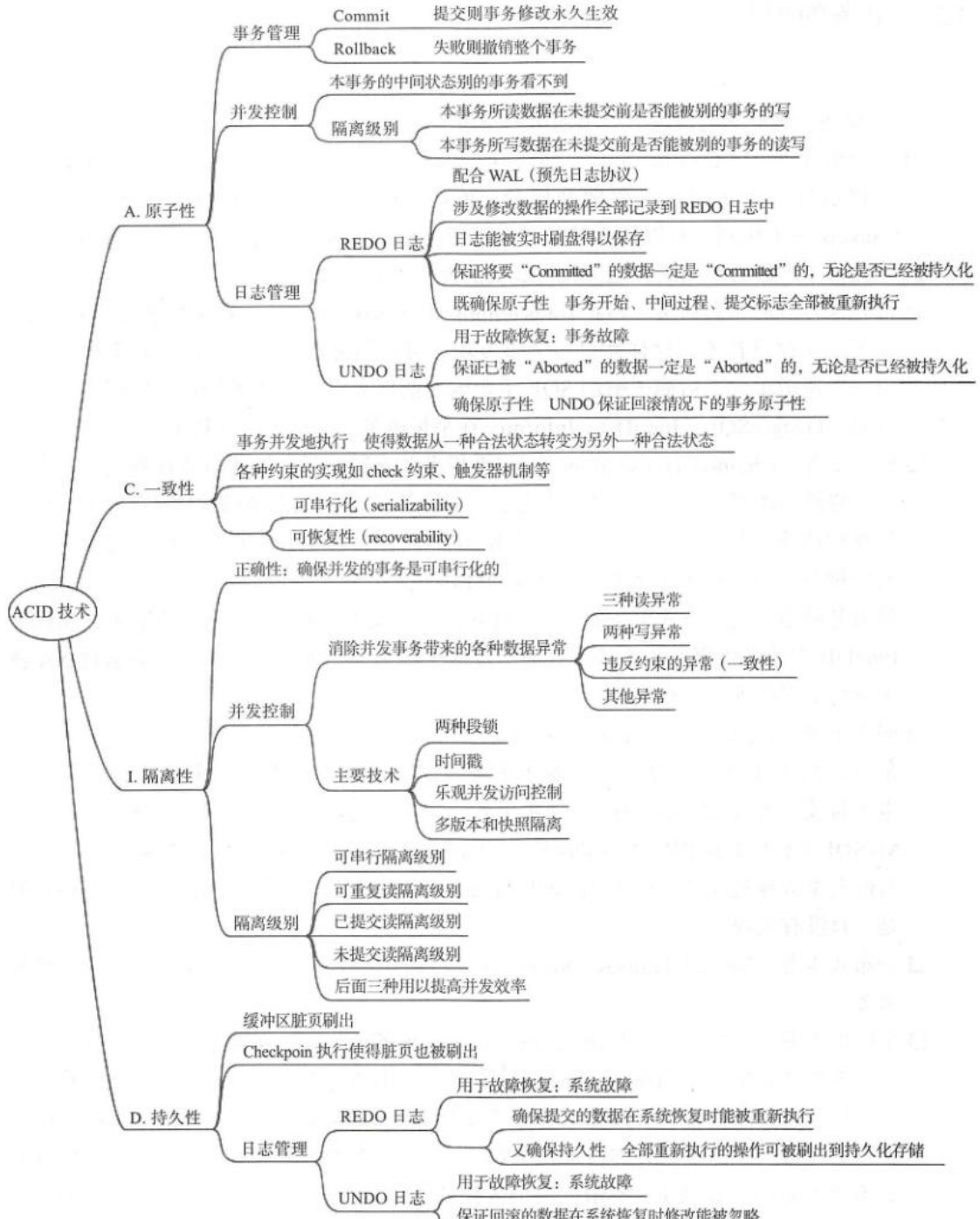
- 针对读读场景可以做优化



事务-MVCC


事务—MVCC
• 本质来说就是copy on write
• 能够做到写不阻塞读
只读事务
读写事务
读写事务
读写事务
读写事务
读写事务
只读事务
只读事务
事务单元1
事务单元2
(2,3单元共享数据)
事务单元3

```
------------
TRANSACTIONS
------------
Trx id counter 24325
Purge done for trx's n:o < 24324 undo n:o < 0 state: running but idle
History list length 3
LIST OF TRANSACTIONS FOR EACH SESSION:
---TRANSACTION 283675367533984, not started
0 lock struct(s), heap size 1136, 0 row lock(s)
---TRANSACTION 283675367535680, not started
0 lock struct(s), heap size 1136, 0 row lock(s)
---TRANSACTION 283675367534832, not started
0 lock struct(s), heap size 1136, 0 row lock(s)
---TRANSACTION 283675367532288, not started
0 lock struct(s), heap size 1136, 0 row lock(s)
--------
```


ACID 技术
A. 原子性
事务管理
Commit
提交则事务修改永久生效
Rollback
失败则撤销整个事务
并发控制
本事务的中间状态别的事务看不到
隔离级别
本事务所读数据在未提交前是否能被别的事务的写
本事务所写数据在未提交前是否能被别的事务的读写
日志管理
REDO 日志
配合 WAL（预先日志协议）
涉及修改数据的操作全部记录到 REDO 日志中
日志能被实时刷盘得以保存
保证将要“Committed”的数据一定是“Committed”的，无论是否已经被持久化
既确保原子性　事务开始、中间过程、提交标志全部被重新执行
UNDO 日志
用于故障恢复：事务故障
保证已被“Aborted”的数据一定是“Aborted”的，无论是否已经被持久化
确保原子性　UNDO 保证回滚情况下的事务原子性
C. 一致性
事务并发地执行　使得数据从一种合法状态转变为另外一种合法状态
各种约束的实现如 check 约束、触发器机制等
可串行化（serializability）
可恢复性（recoverability）
I. 隔离性
正确性：确保并发的事务是可串行化的
并发控制
消除并发事务带来的各种数据异常
三种读异常
两种写异常
违反约束的异常（一致性）
其他异常
主要技术
两种段锁
时间戳
乐观并发访问控制
多版本和快照隔离
隔离级别
可串行隔离级别
可重复读隔离级别
已提交读隔离级别
未提交读隔离级别
后面三种用以提高并发效率
D. 持久性
缓冲区脏页刷出
Checkpoin 执行使得脏页也被刷出
日志管理
REDO 日志
用于故障恢复：系统故障
确保提交的数据在系统恢复时能被重新执行
又确保持久性　全部重新执行的操作可被刷出到持久化存储
UNDO 日志
用于故障恢复：系统故障

事务实现原理

- 1 redo
- 2 undo
- 3 purge
- 4 group commit

- A：redo：记录事务行为
- C：undo：事务回滚-Rollback，MVCC
- I：Lock：共享锁和互斥锁，行锁，表锁，record-lock，gap-lock，next-key lock，插入意向锁，AUTO-INC locks
- D：redo

事务语句

- START TRANSACTION;
- COMMIT;
- SAVEPOINT;
- ROLLBACK;

- 隐式提交：
  - DDL
  - CREATE USER，DROP USER，GRANT，RENAME USER
  - ANALYZE TABLE，OPTIMIZE TABLE

在看隔离性 I

- 隔离性
  - 1 正确性(happen before)
  - 2 并发控制(Lock)
  - 3 隔离性(SQL92标准四种)

隔离性探索

隔离级别	脏　读	不可重复读	幻　象　读	第一类丢失更新	第二类丢失更新
READ UNCOMMITED	允许	允许	允许	不允许	允许
READ COMMITTED	不允许	允许	允许	不允许	允许
REPEATABLE READ	不允许	不允许	允许	不允许	不允许
SERIALIZABLE	不允许	不允许	不允许	不允许	不允许

Repeatble Read的Next-key Lock解决不可重复读问题，已经到了SQL92的SERILIZABLE标准，这张图对吗？

更全的隔离性

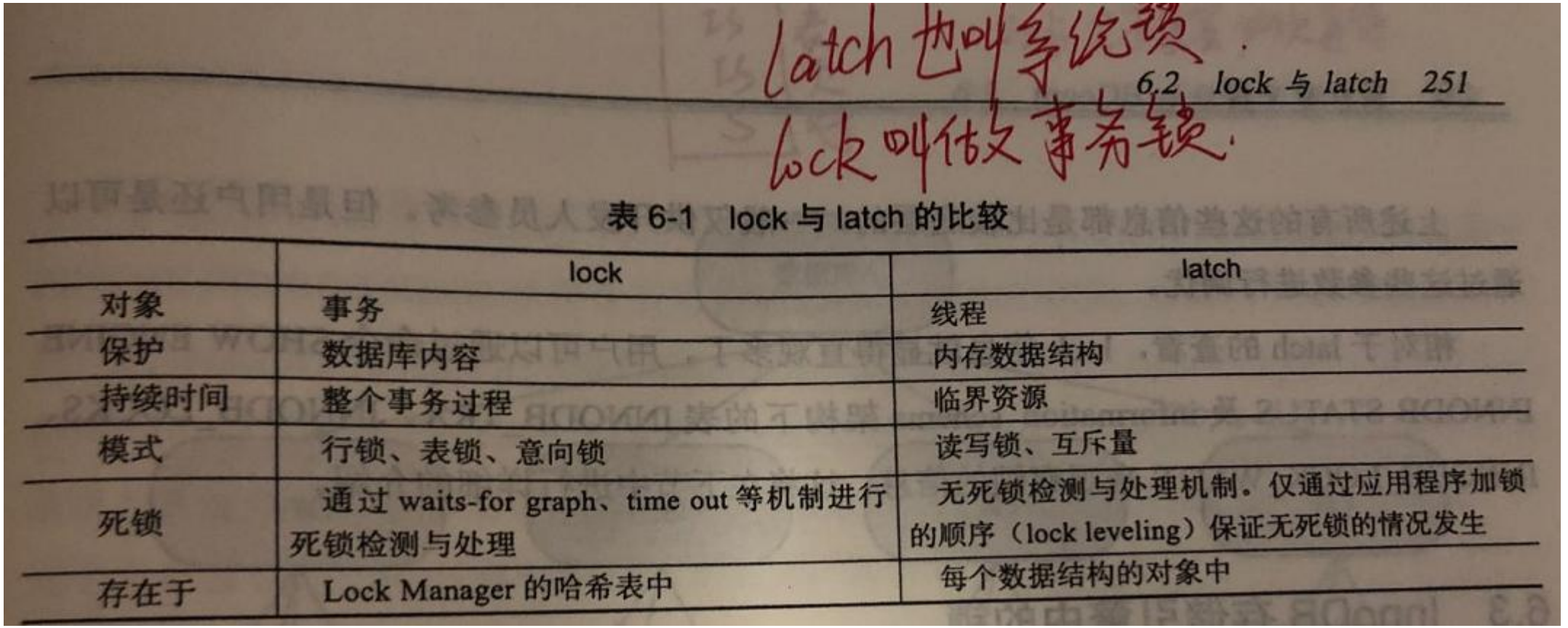
Table 4. Isolation Types Characterized by Possible Anomalies Allowed.

Isolation level	**P0** Dirty Write	**P1** Dirty Read	**P4C** Cursor Lost Update	**P4** Lost Update	**P2** Fuzzy Read	**P3** Phantom	**A5A** Read Skew	**A5B** Write Skew
READ UNCOMMITTED == Degree 1	Not Possible	Possible	Possible	Possible	Possible	Possible	Possible	Possible
READ COMMITTED == Degree 2	Not Possible	Not Possible	Possible	Possible	Possible	Possible	Possible	Possible
Cursor Stability	Not Possible	Not Possible	Not Possible	Sometimes Possible	Sometimes Possible	Possible	Possible	Sometimes Possible
REPEATABLE READ	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible	Possible	Not Possible	Not Possible
Snapshot	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible	Sometimes Possible	Not Possible	Possible
ANSI SQL SERIALIZABLE == Degree 3 == Repeatable Read Date, IBM, Tandem, ...	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible

隔离性实例

Application	Isolation
SQLite	Strong isolation
MariaDB	Four isolation levels in SQL standards [38]
Kyoto Cabinet	Intentionally no isolation
APT	No isolation
vim	Strong isolation or no isolation

Lock And Latch



latch也叫系统锁.
lock叫做事务锁.

表 6-1 lock 与 latch 的比较

	lock	latch
对象	事务	线程
保护	数据库内容	内存数据结构
持续时间	整个事务过程	临界资源
模式	行锁、表锁、意向锁	读写锁、互斥量
死锁	通过 waits-for graph、time out 等机制进行死锁检测与处理	无死锁检测与处理机制。仅通过应用程序加锁的顺序（lock leveling）保证无死锁的情况发生
存在于	Lock Manager 的哈希表中	每个数据结构的对象中

1 脏读

时　间	转账事务 A	取款事务 B
T1		开始事务
T2	开始事务	
T3		查询账户余额为 1000 元
T4		取出 500 元，把余额改为 500 元
T5	查询账户余额为 500 元（脏读）	
T6		撤销事务，余额恢复为 1000 元
T7	汇入 100 元，把余额改为 600 元	
T8	提交事务	

2 不可重复读

时　间	取款事务 A	转账事务 B
T1		开始事务
T2	开始事务	
T3		查询账户余额为 1000 元
T4	查询账户余额为 1000 元	
T5		取出 100 元，把余额改为 900 元
T6		提交事务
T7	查询账户余额为 900 元（和 T4 读取的不一致）	

3 幻读

时　间	统计金额事务 A	转账事务 B
T1		开始事务
T2	开始事务	
T3	统计总存款数为 10000 元	
T4		新增一个存款账户，存款为 100 元
T5		提交事务
T6	再次统计总存款数为 10100 元（幻象读）	

4 脏写(第一类丢失更新)

时　间	取款事务 A	转账事务 B
T1	**开始事务**	
T2		**开始事务**
T3	查询账户余额为 1000 元	
T4		查询账户余额为 1000 元
T5		汇入 100 元，把余额改为 1100 元
T6		**提交事务**
T7	取出 100 元，把余额改为 900 元	
T8	**撤销事务**	
T9	余额恢复为 1000 元（丢失更新）	

5 第二类丢失更新

时　间	转账事务 A	取款事务 B
T1		开始事务
T2	开始事务	
T3		查询账户余额为 1000 元
T4	查询账户余额为 1000 元	
T5		取出 100 元，把余额改为 900 元
T6		提交事务
T7	汇入 100 元	
T8	提交事务	
T9	把余额改为 1100 元（丢失更新）	

https://blog.51cto.com/thinklili/2500781

5 Optimization

- 5.1 问题驱动：根据问题分析MySQL
- 5.2 体系结构驱动：
  - Schema和数据类型优化
  - 索引优化
  - 查询性能优化
  - Mysql服务器优化
  - 操作系统和硬件优化
  - 应用优化
- 5.3 官方优化指南(强烈推荐)
  - https://dev.mysql.com/doc/refman/8.0/en/optimize-overview.html

5.3 官方优化指南

8.2 Optimizing SQL Statements

8.3 Optimization and Indexes

8.4 Optimizing Database Structure

8.5 Optimizing for InnoDB Tables

8.8 Understanding the Query Execution Plan

8.10 Buffering and Caching

MySQL uses several strategies that cache information in memory buffers to increase performance.