SQL Server内存数据库揭秘

DTCC

2015中国数据库技术大会

DATABASE TECHNOLOGY CONFERENCE CHINA 2015

大数据技术探索和价值发现

About me

易车 yiche.com 买车无难事

@shanksgao

博客园 cnblogs.com

agenda

为什么需要全新的引擎？

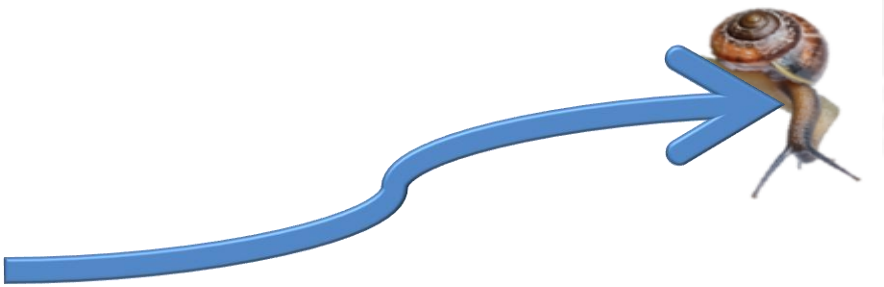
Hekaton实现

带来的帮助与挑战

RDBMS的发展

内存价格日趋便宜 低于10$/GB
摩尔定律魔力减弱,CPU计算提速减慢
多核架构发展迅速

传统数据库经过多年发展如今性能提升缓慢



100倍速度提升?

现有数据库引擎木有可能。。

假设

所有表均在内存中

100万指令/事务　100 TPS

10X

1,000 TPS?

10万指令/事务

90% CPU指令减少

100X

10,000 TPS?

1万指令/事务

– 99% CPU指令减少

为什么要新引擎？

现有引擎中将所有表数据装入内存后我们还会面临。。。

内存中的共享数据架构所使用的闩锁(热区问题)

为保证并发事务所采用的锁技术(阻塞/并发问题)

当前数据库引擎执行方式(语句执行效率,CPU高消耗)

HEKATON

基于内存优化,但持久化

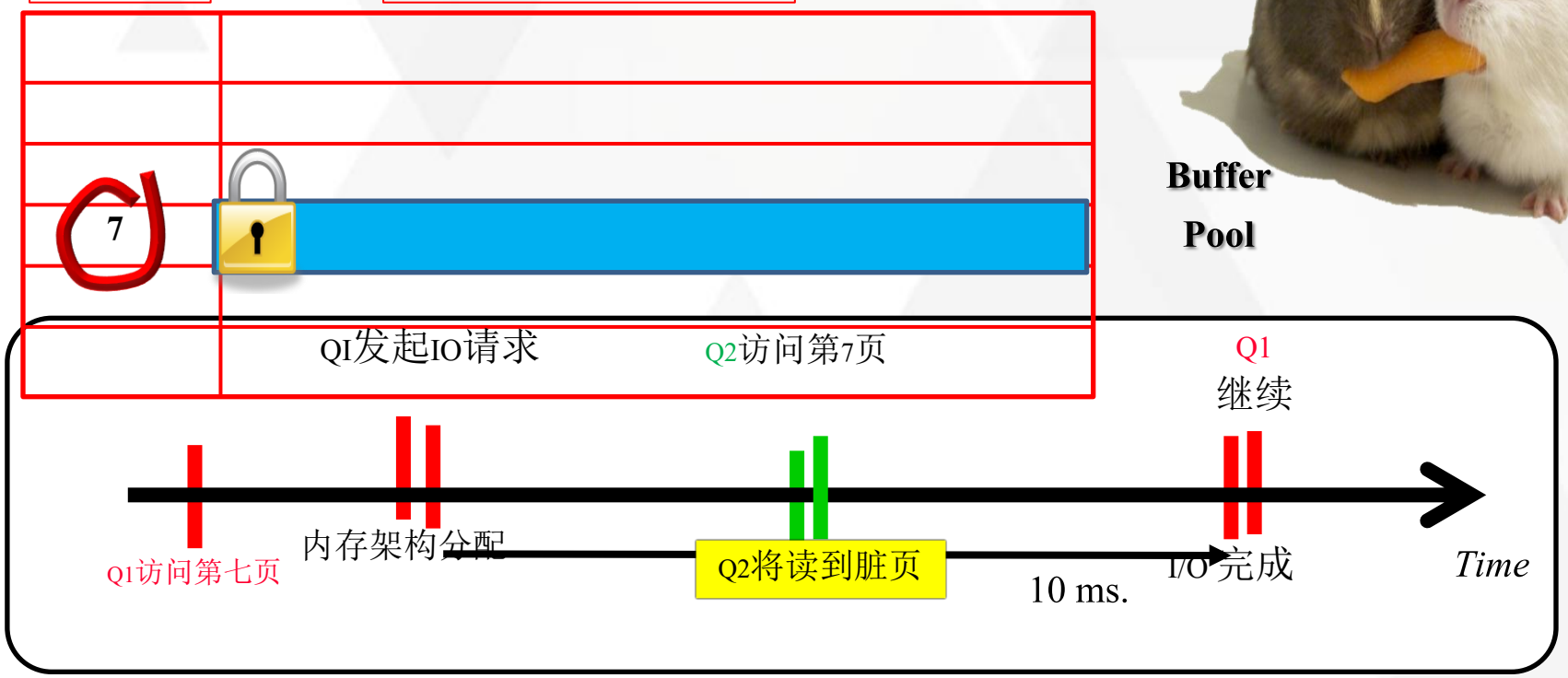
高性能的OLTP引擎

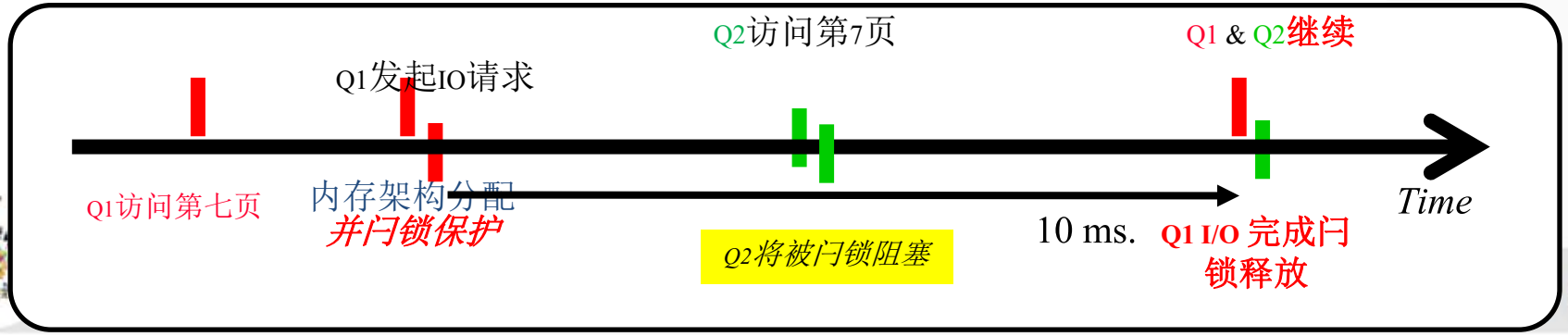
完全集成于现有SQL Server中

基于当前CPU架构设计

共享缓冲池中闩锁(Buffer Pool)

Buffer Pool

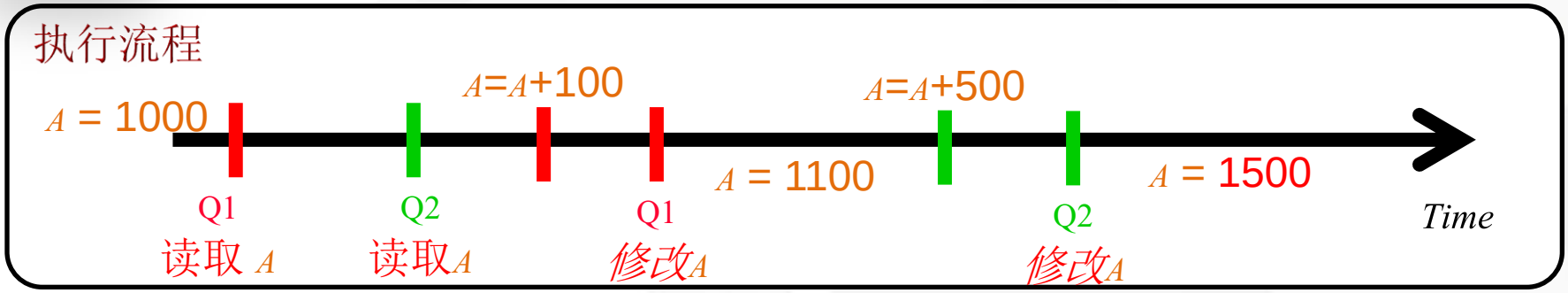


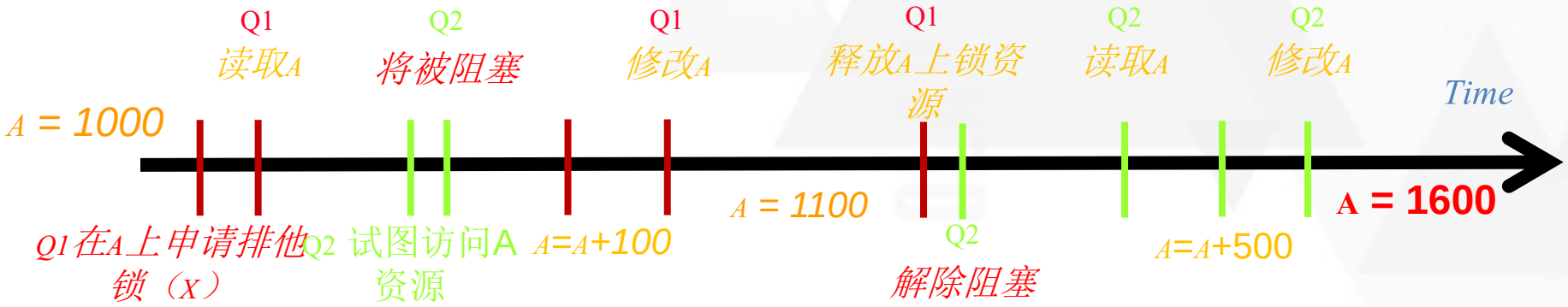


Jim Gray –两阶段锁

Q1: $A = A + 100$ DB 操作: Read A, $A=A+100$, Write A

Q2: $A = A + 500$ DB 操作: Read A, $A=A+500$, Write A





解释性语言(执行计划)

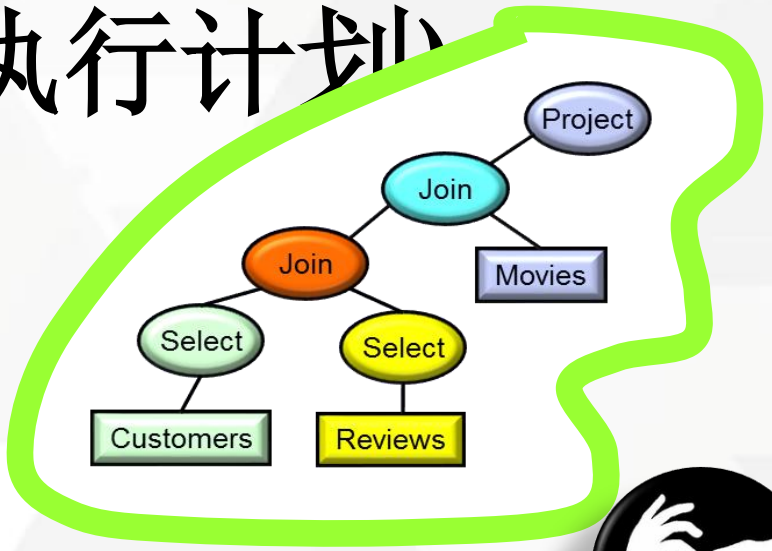
Select…From…Where…

语义分析，优化后生成SQL执行计划

执行相应计划时

调用数据库的函数，运行每个运算符

当数据在硬盘上时计算资源/时间高消耗



Expensive

当数据在硬盘上时计算资源/时间高消耗

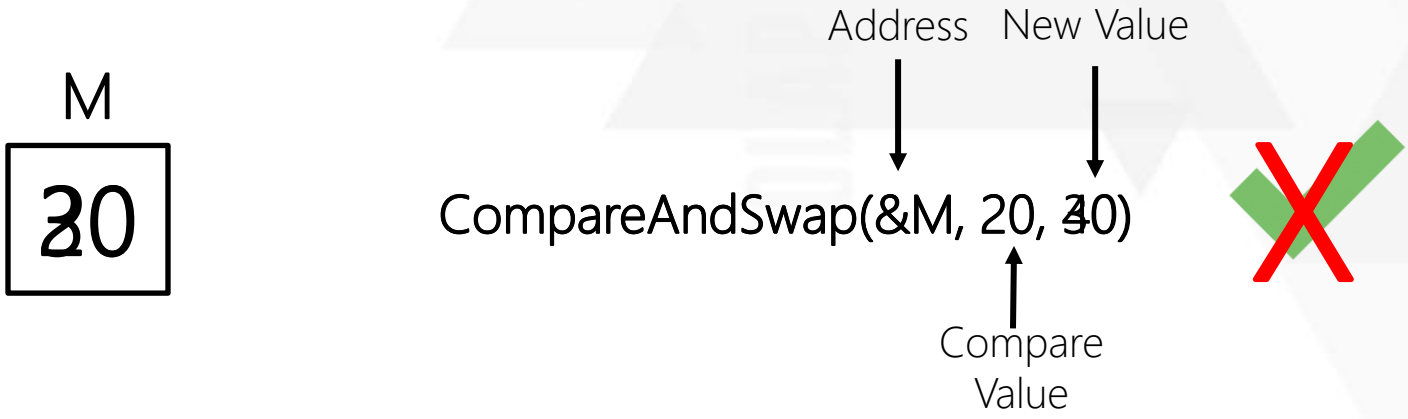
Cheaper

如果数据均在内存中可以改进很多！

DIM	SQL SERVER	HEKATON
Shared data structures	Latches	Lock-free data structures
Concurrency control	Locking	Versions w/ timestamps + optimistic concurrency control
Query Execution	Interpretation	Compilation into DLL

Compare and Swap(CAS)

```
int compare_and_swap(int* reg, int oldval, int newval)
{
  ATOMIC();
  int old_reg_val = *reg;
  if (old_reg_val == oldval)
     *reg = newval;
  END_ATOMIC();
  return old_reg_val;
}
```

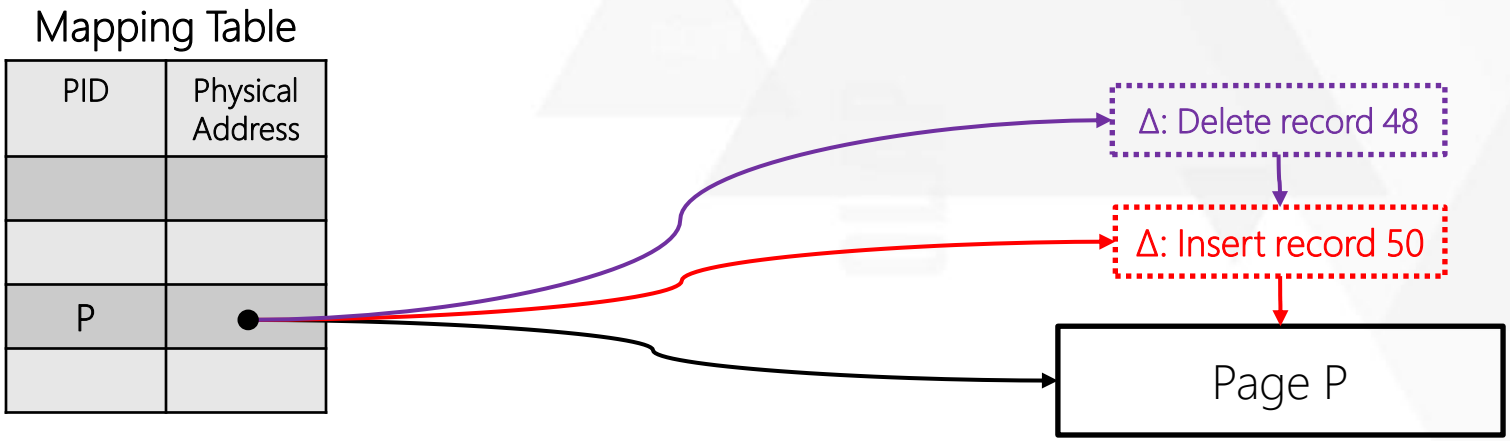


增量更新(Delta Updates)

每次在某一数据页上的更新将生成一个新的地址(Delta)

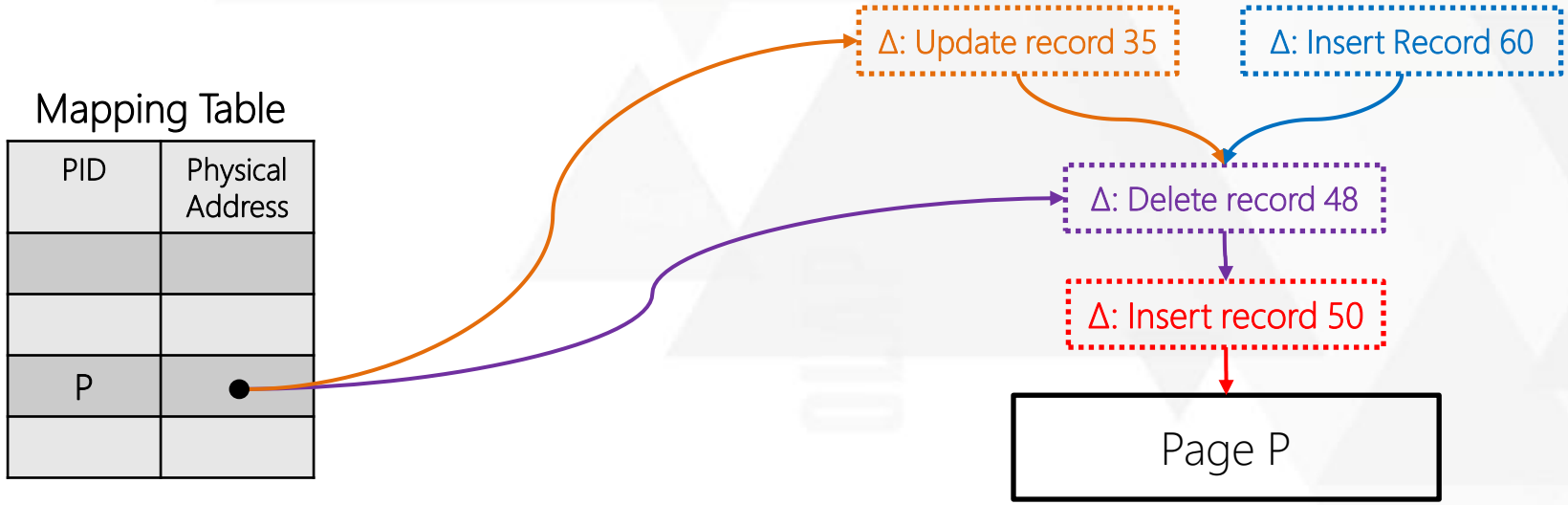
新地址将作为现有数据库页的入口

采用CAS完成映射表中(mapping table)物理新地址的映射(delta address)



更新竞争（Update Contention）

针对同一数据库页可能出现同时更新
此时胜出者更新,失败者需重试.

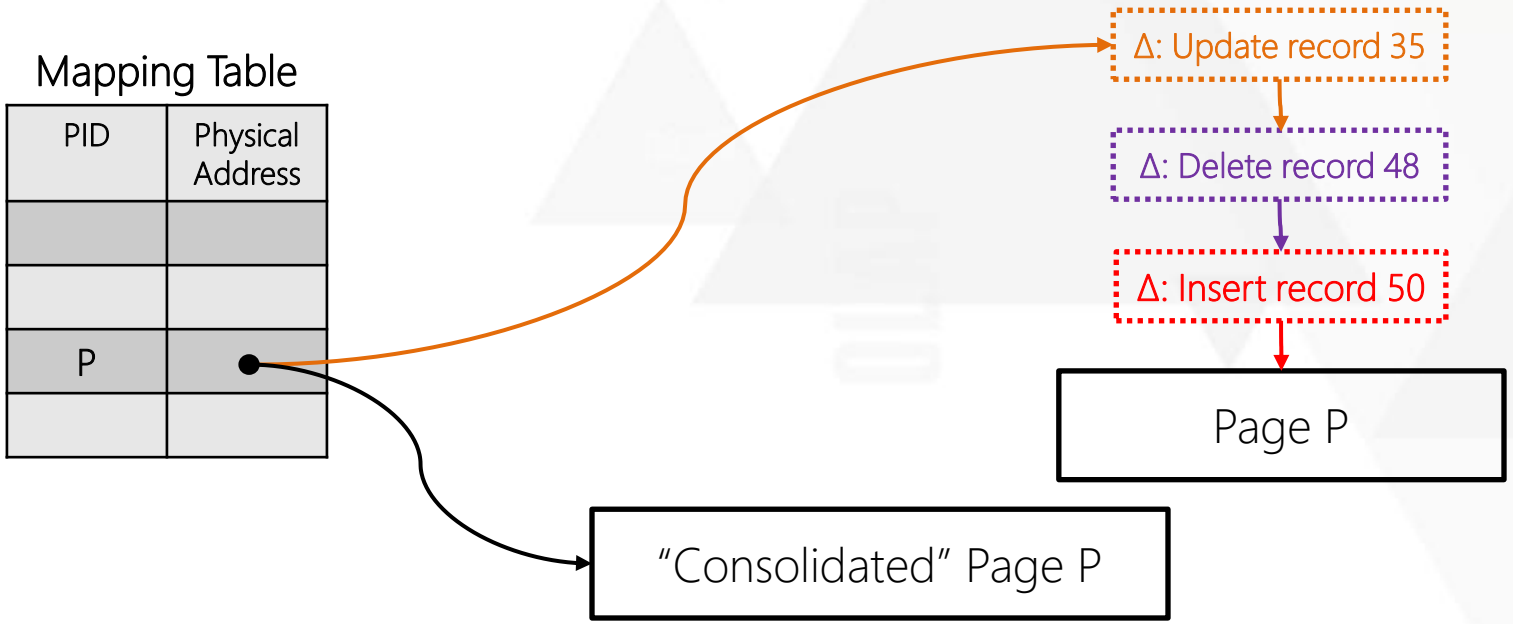


内存中数据页整合

增量更新链造成查询性能下降

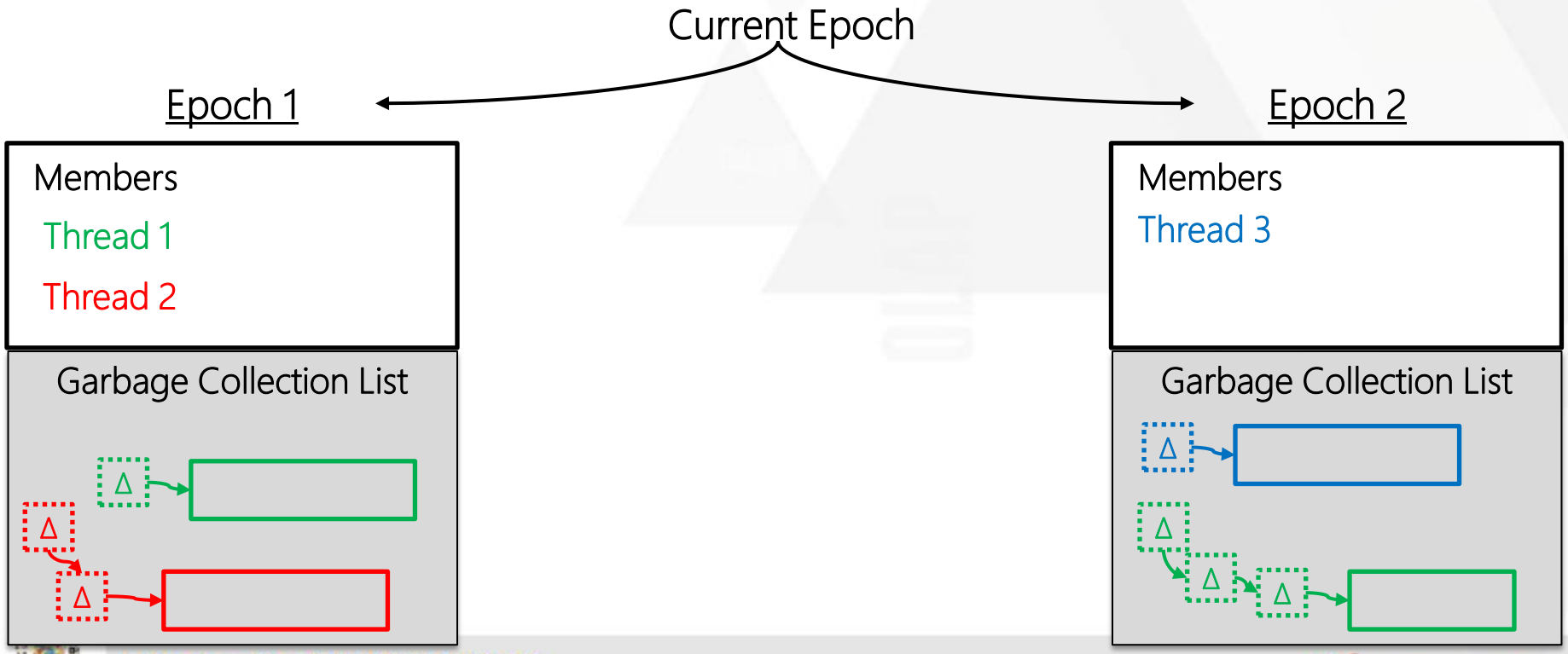
SQL Server会将其整合成新的数据页

旧的数据页将会作为垃圾回收释放内存



垃圾回收

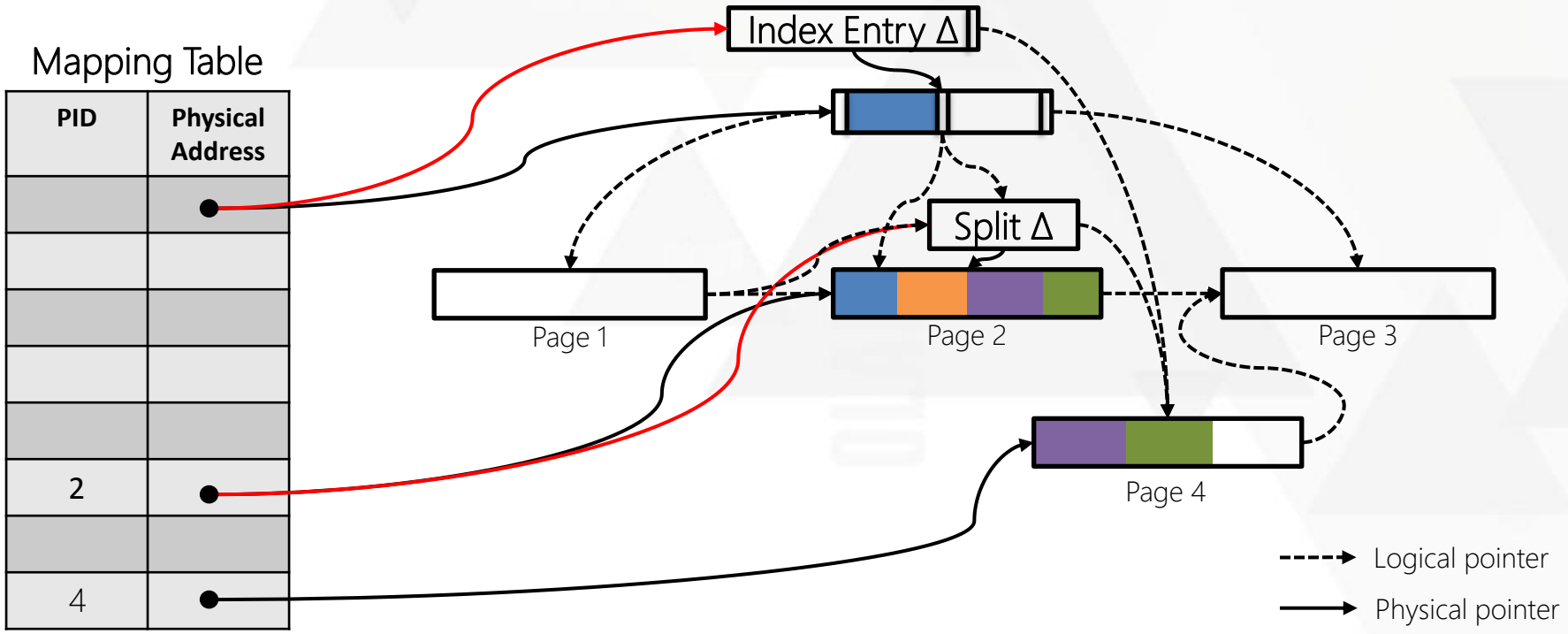
每个线程开始执行操作前都会赋予一个时间点(epoch)
将当前的时间点公示
当所有线程都退出当前的时间点后进行回收



structure modification operations(SMOS)
Latch-free下页分裂

数据页大小是弹性的

无特定的值触发页分裂，合适的状态时间进行



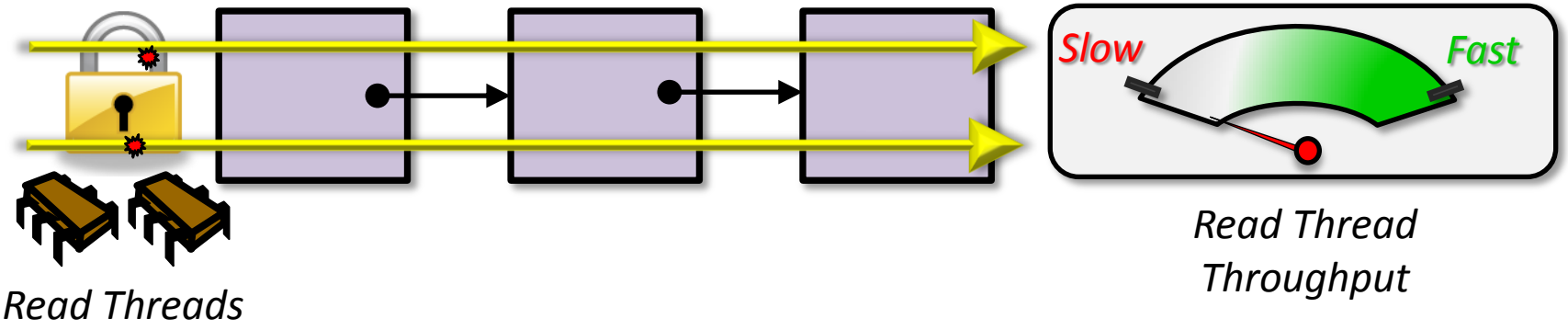
Latch-free “half-split”

1.子节点层创建新的分裂页面

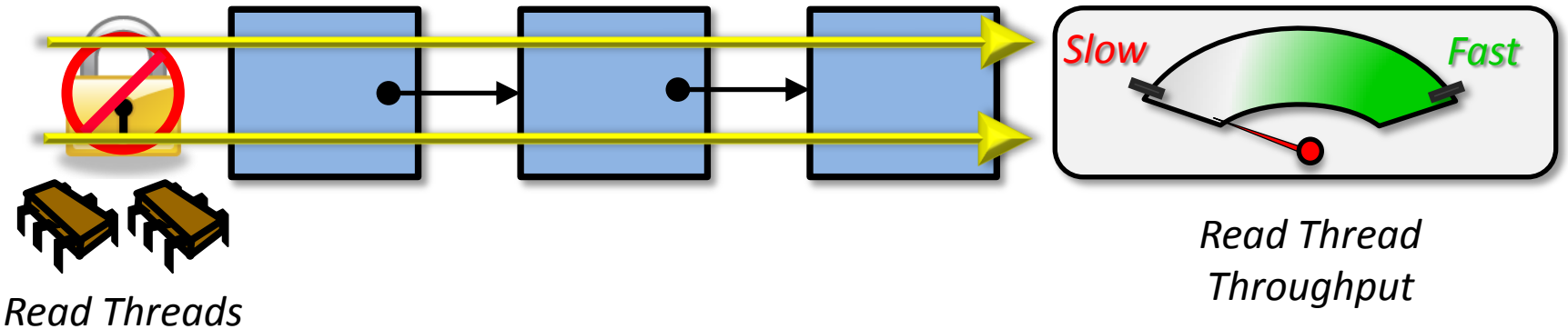
2.载入新的分裂Key并指向父节点层

Latch vs. Lock-Free

Latch

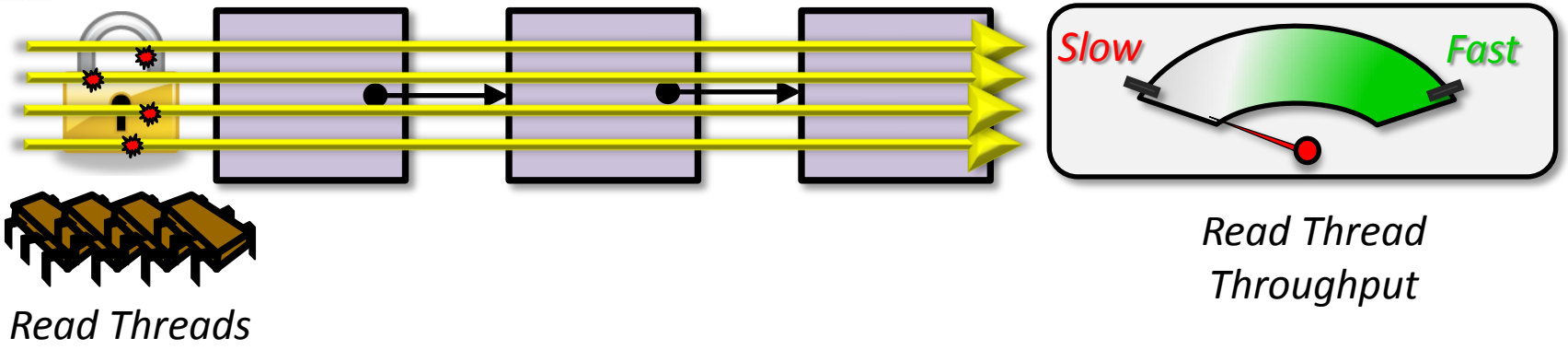


Lock-free

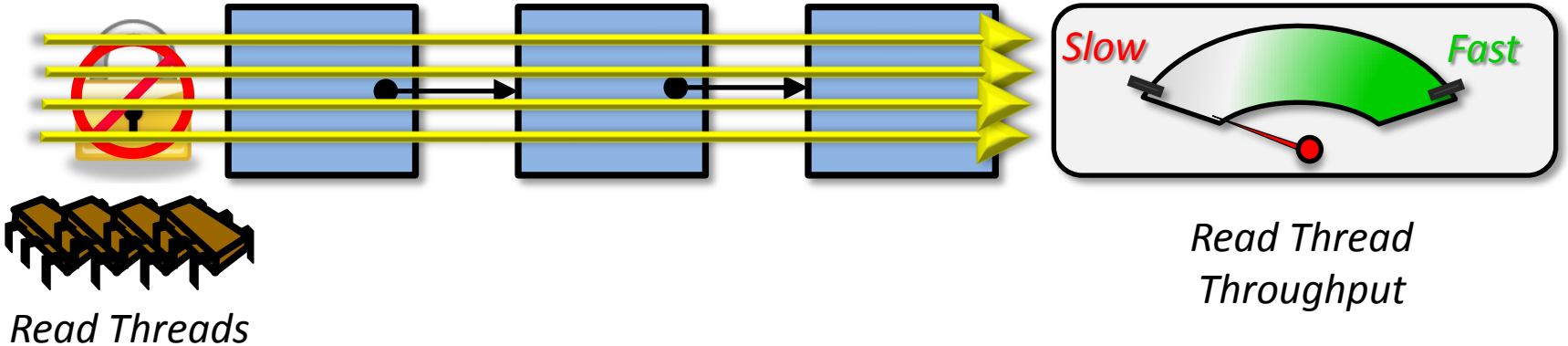


Latch vs. Lock-Free

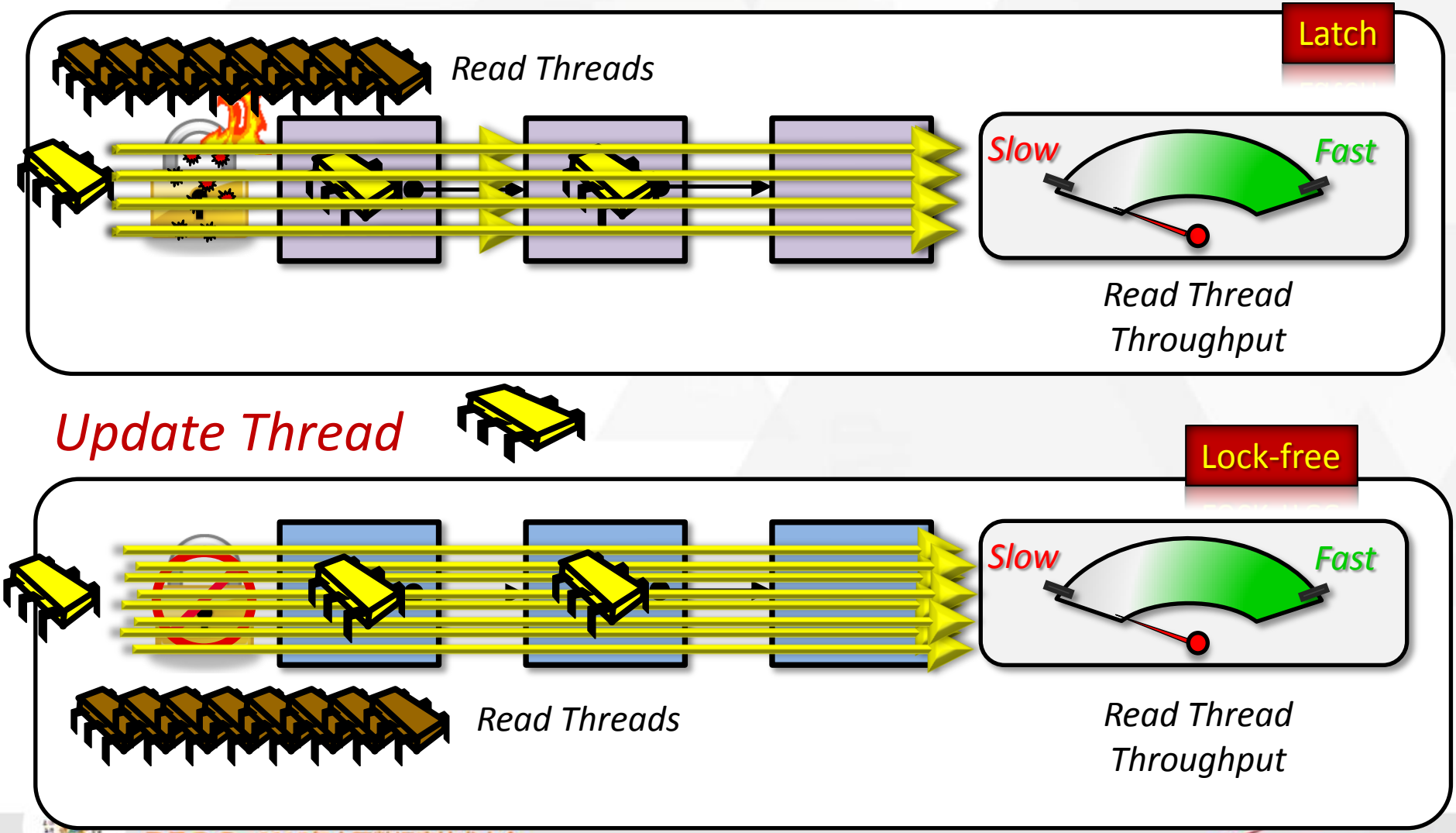
Latch



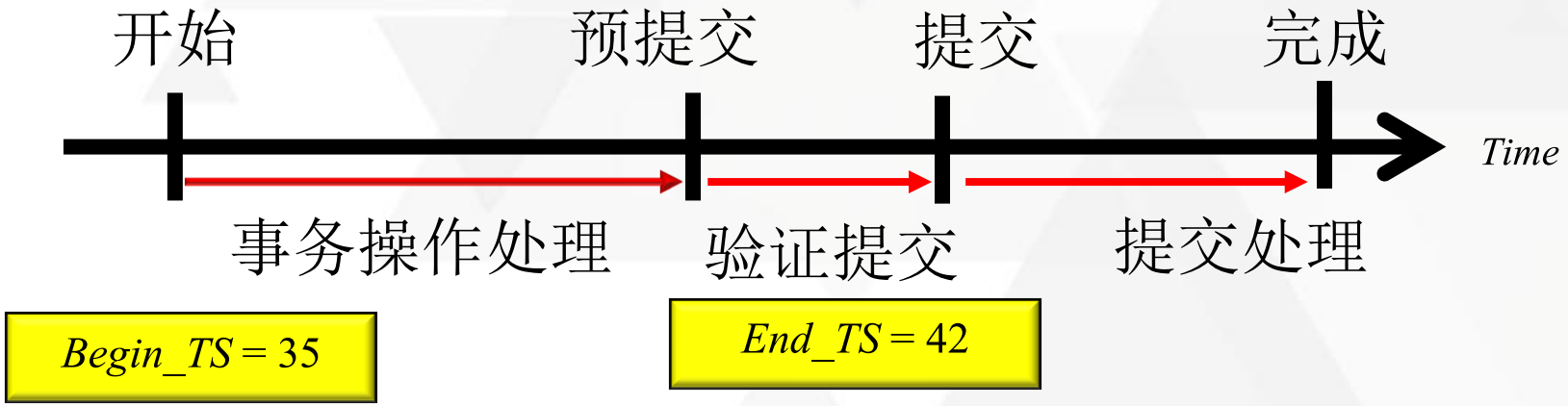
Lock-free



Latch vs. Lock-Free



Hekaton中的事务阶段



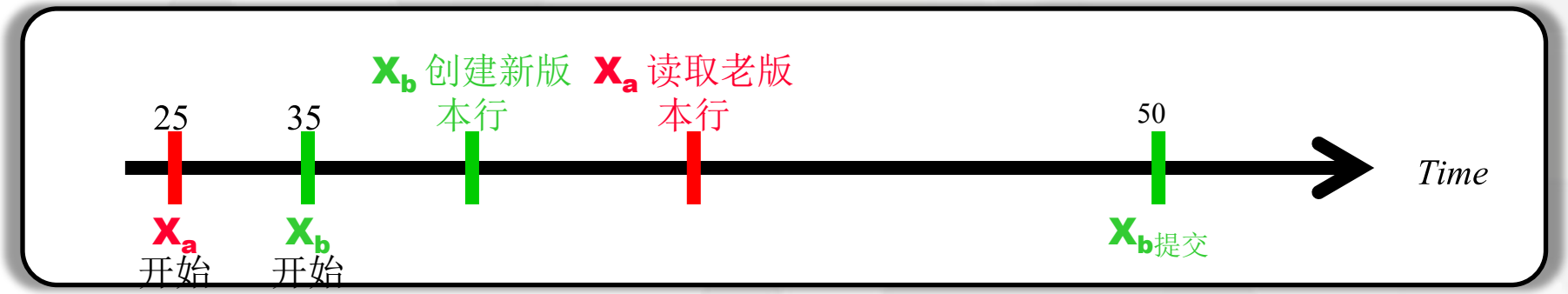
采用Timestamps(时间戳(全局时钟))标记事务和行版本

Begin_TS:事务开始时赋予开始时间戳用于读取正确的行版本
(读取已提交的版本行数据,更新操作创建新的版本行并跟踪读,写数据集 用于判断)

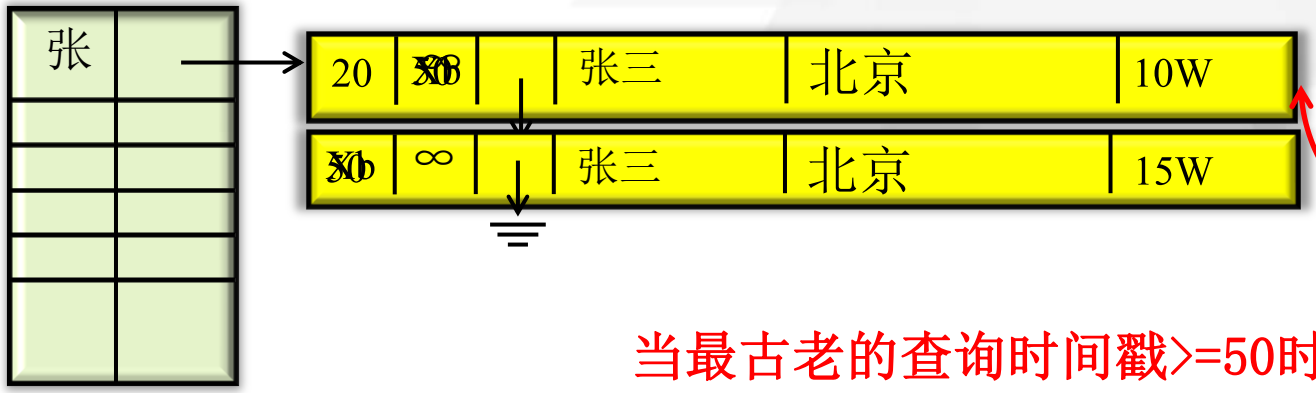
End_TS:验证提交时赋予结束时间戳,并用于判断是否可以安全提交(根据隔离级别)

提交处理: WAL,使新版本数据对其他事务可见

版本行并发控制



数据表(哈希)



当最古老的查询时间戳>=50时，垃圾回收

分析

- 事务X_a 开始,为其分配开始时间戳 *Begin_TS* 25
- 事务 X_b 开始,为其分配开始时间戳 *Begin_TS* 35
  - （*意味着X_b结束时间戳End_TS* of X_b > 35）
- 当 X_a 检索到此条记录时

- 此时取决于X_b's开始时间戳 *Begin_TS* (35)

由于X_a开始时间戳 小于X_b开始时间戳， 所以读取正确

分析

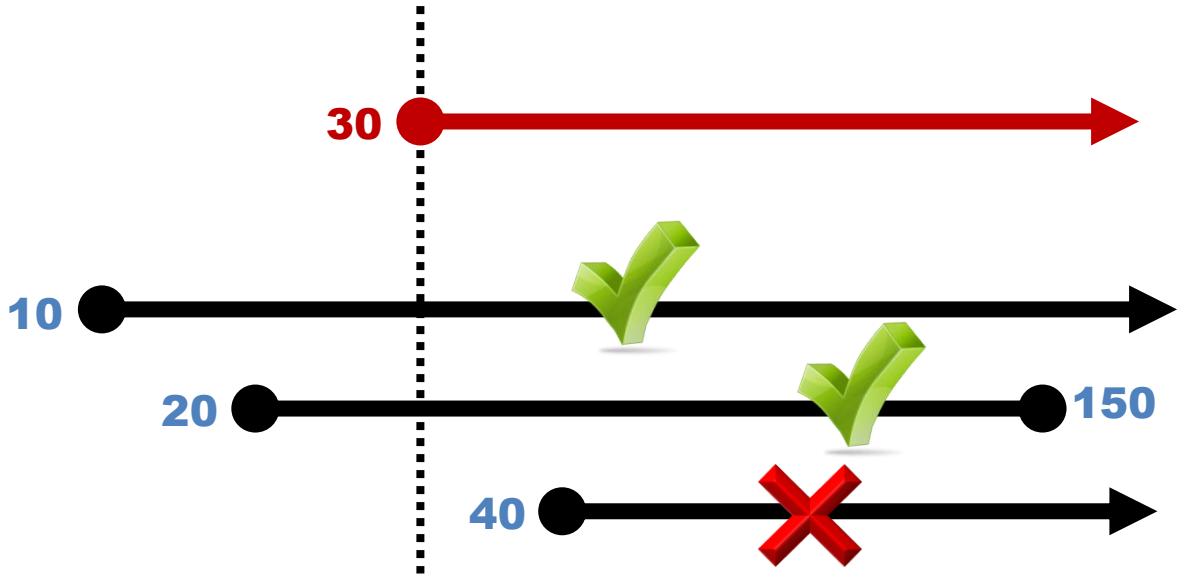
每个行版本数据都有一个时间区间

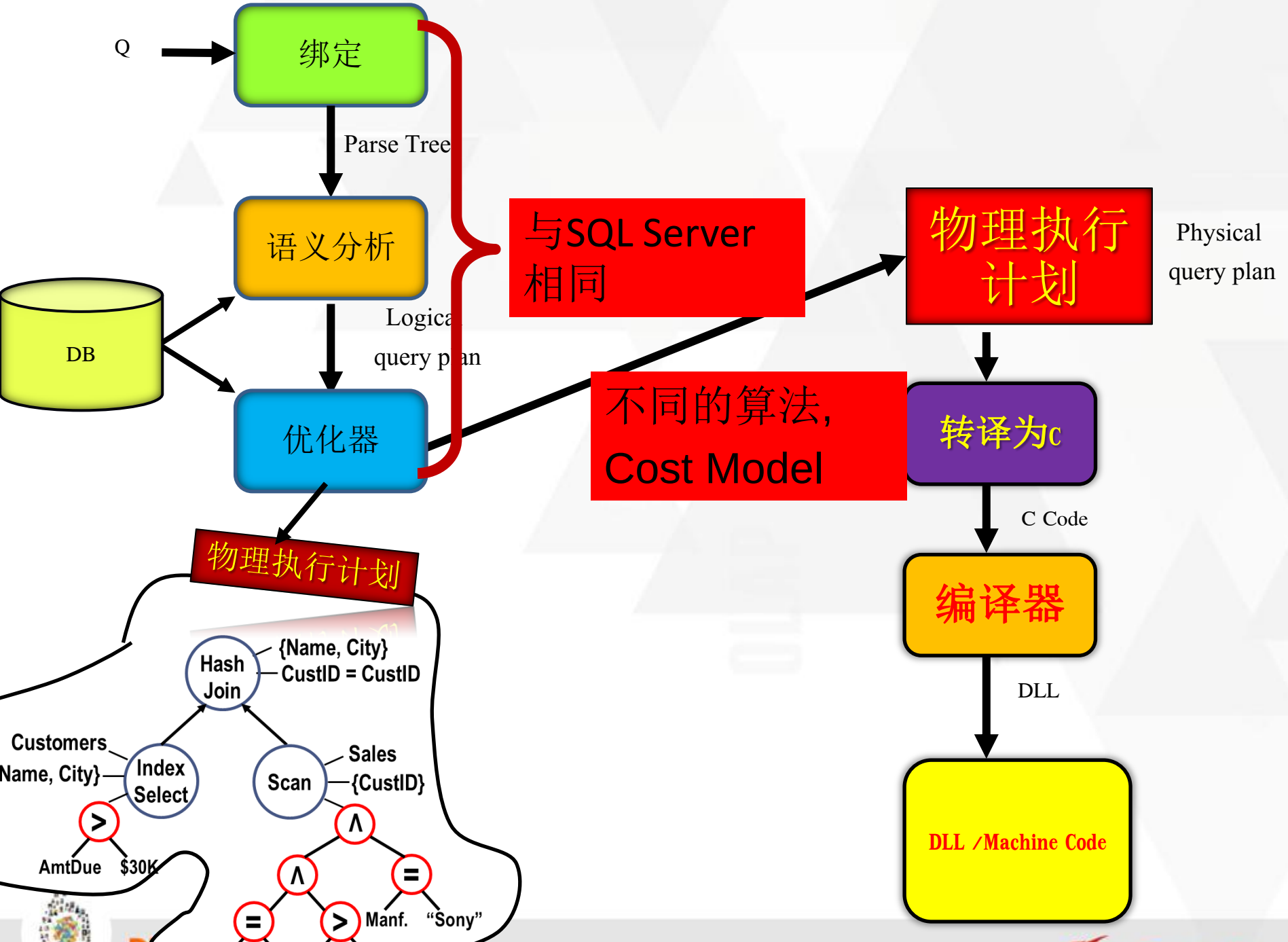
查询的可见值区间必须覆盖此查询的开始时间戳

例:

Xa的Begin_TS为30

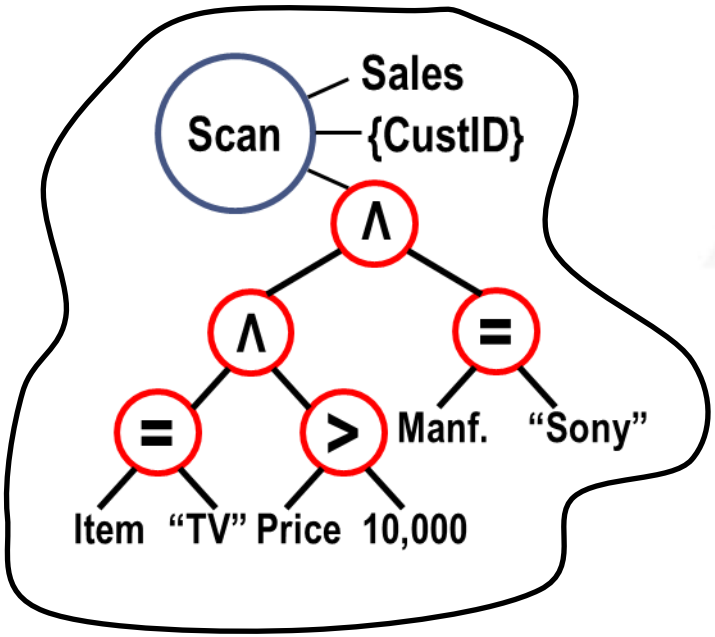
Xa的可见值区间为




Q
绑定
Parse Tree
语义分析
DB
Logical
query plan
优化器
与SQL Server
相同
不同的算法,
Cost Model
物理执行
计划
Physical
query plan
转译为c
C Code
编译器
DLL
DLL /Machine Code
物理执行计划
Hash
Join
{Name, City}
CustID = CustID
Customers
Name, City}
Index
Select
Scan
Sales
{CustID}
>
AmtDue
$30K
∧
∧
=
=
>
Manf.
"Sony"

“Native” vs. “Interpreted”

谓词检索



HK “interop” case

	Interpreted QP		Native QP
	SQL Table	Hekaton Table	Hekaton Table
Matching Row	700	332	75
Non-matching Row	300	110	31

每行数据消耗的指令数

查询效率

随机查找1000W行的表

所有数据在内存中

Intel Xeon W3520 2.67 GHz

Transaction size in #lookups	CPU cycles (in millions)		Speedup
	SQL Table	Hekaton Table	
1	0.734	**0.040**	**10.8X**
10	0.937	**0.051**	**18.4X**
100	2.72	**0.150**	**18.1X**
1,000	20.1	**1.063**	**18.9X**
10,000	201	**9.85**	**20.4X**

Hekaton 性能: 270W lookups/sec/core

更新效率

随机更新，1000W行数据，一个索引，快照隔离级别
关闭写日志(否则磁盘将成为瓶颈)
Intel Xeon W3520 2.67 GHz

Transaction size in #updates	CPU cycles (in millions)		Speedup
	SQL Table	Hekaton Table	
1	0.910	**0.045**	**20.2X**
10	1.38	**0.059**	**23.4X**
100	8.17	**0.260**	**31.4X**
1,000	41.9	**1.50**	**27.9X**
10,000	439	**14.4**	**30.5X**

Hekaton 性能：190W updates/sec/core

案例

易车·惠买车

底价自然来

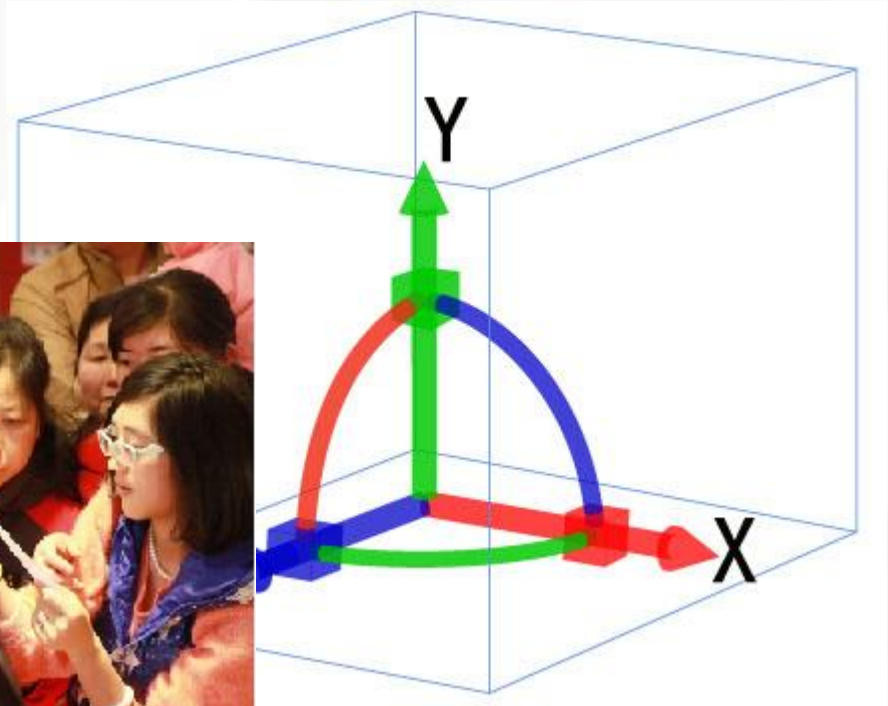
AVG 12,000 requests/sec

MAX 30000+ requests/sec

秒杀

计算机的当下的普遍应用

模拟重现三维空间内活动



激动的特性完美无缺?

任何术都是有缺陷的

没有哪项技术是完美无缺的.

合适的场景应用合理的技术.


ChinaUnix
ITPUB
THANKS