Hekaton

Tidying up SQL Server

Cristian Diaconu, et. al.
SIGMOD 2013

Michael Abebe
CS 848 (January 2018)
the life-changing magic of tidying up

the Japanese art of decluttering and organizing

marie kondo
Discard anything that does not bring you joy.
2006 Databases*

Threads  locks  Buffer pool  disk
1970s Databases

Designed to **mask disk latency**

- **Threads**
- **locks**
- **Buffer pool**
- **disk**
1970s Hardware

~1 MB

~100 ms seek
2006 Hardware

- Memory: 10-100 GB
- Hard drive: 10,000 x faster
- Seek time: ~10 ms
- 10 x faster
2006 Databases*

10-100 GB

50 byte records
120 months
10 million users =
60 GB

Data fits in memory
Discard anything that does not bring you joy
Increased contention

Threads  Locks  Buffer pool  Disk
How to ensure correctness?

Threads  Buffer pool
Discarding Everything

The End of an Architectural Era
(It’s Time for a Complete Rewrite)

OLTP Through the Looking Glass, and What We Found There

Stavros Harizopoulos
HP Labs
Palo Alto, CA
stavros@hp.com

Daniel J. Abadi
Yale University
New Haven, CT
dna@cs.yale.edu

Samuel Madden
Massachusetts Institute of Technology
Cambridge, MA
{madden, stonebraker}@csail.mit.edu

ABSTRACT

Online Transaction Processing (OLTP) databases include a suite of features — disk-resident B-trees and heap files, locking-based concurrency control, support for multi-threading — that were optimized for computer technology of the late 1970’s. Advances in modern processors, memories, and networks mean that today’s computers are vastly different from those of 30 years ago, such that many OLTP databases will now fit in main memory, and most OLTP transactions can be processed in milliseconds or less. Yet database architecture has changed little.

Based on this observation, we look at some interesting variants of

1. INTRODUCTION

Modern general purpose online transaction processing (OLTP) database systems include a standard suite of features: a collection of on-disk data structures for table storage, including heap files and B-trees, support for multiple concurrent queries via locking-based concurrency control, log-based recovery, and an efficient buffer manager. These features were developed to support transaction processing in the 1970’s and 1980’s, when an OLTP database was many times larger than the main memory, and when the computers that ran these databases cost hundreds of thousands to millions of dollars.
Partitioned Execution

Execute serially

Partition

Execution thread per partition
Partitioned Execution

\[ \text{Tput} = (1 \text{ Core Tput}) \times (# \text{ Cores}) \]
Partitioned Execution

Multi-partition transactions?

Costly coordination
Partitioned Execution

Multi-partition transactions?

\[ \text{Tput} = ( \text{1 Core Tput} ) \times \text{Scalability}^{(\text{# Cores})} \]

Based on partition quality
How to improve throughput?

\[ \text{Tput} = (1 \text{ Core Tput}) \times \text{Scalability} \times \left( \frac{\# \text{ Cores}}{1} \right) \]

- Eliminate instructions
- Eliminate contention
- Eliminate locks
Hekaton

- Compiler
  - Eliminate instructions
- Runtime
  - Eliminate locks
- Storage Engine
Discard anything that does not bring you joy unless it makes you money.
Hekaton in SQL Server

Compiler
Runtime
Storage Engine

Metadata
Optimizer
Processor
Storage
Hekaton

Compiler  
Eliminate instructions

Runtime  
Eliminate locks

Storage Engine
Indexes

Lock free: Hash Table and B-Tree

The Bw-Tree: A B-tree for New Hardware Platforms

Building a Bw-Tree Takes More Than Just Buzz Words

Ziqi Wang
Carnegie Mellon University
ziqiw@cs.cmu.edu

Andrew Pavlo
Carnegie Mellon University
pavlo@cs.cmu.edu

Hyeontaek Lim
Carnegie Mellon University
hl@cs.cmu.edu

Viktor Leis
TU München
leis@in.tum.de

Huanchen Zhang
Carnegie Mellon University
huanchen1@cs.cmu.edu

Michael Kaminsky
Intel Labs
michael.e.kaminsky@intel.com

David G. Andersen
Carnegie Mellon University
dga@cs.cmu.edu

Abstract—The emergence of new hardware platforms led to reconsideration of how data management systems are designed. However, certain access to records remains an architectural layering architectural design decisions about the Bw-tree achieves its very high performance via a latch-free approach that effectively multi-core chips. Our structuring that blurs the store and works well with architecture and algorithms memory aspects. The paper that demonstrate that performance.

ABSTRACT

In 2013, Microsoft Research proposed the Bw-Tree (humorously termed the "Buzz Word Tree"), a lock-free index that provides high throughput for transactional database workloads in SQL Server’s Hekaton engine. The Bw-Tree avoids locks by appending delta record to tree nodes and using an indirection layer that allows it to atomically update physical pointers using compare-and-swap (CAS). Correctly implementing these techniques requires careful attention to detail. Unfortunately, the Bw-Tree papers from Microsoft are missing important details and the source code has not been released. This paper has two contributions: First, it is the missing guide for how to build a lock-free Bw-Tree. We clarify missing points in Microsoft’s original design documents and then present techniques to improve the index’s performance. Although our focus here is on usually not explicitly stated in the serial version of the algorithm. Programmers often implement lock-free algorithms incorrectly and end up with busy-waiting loops. Another challenge is that lock-free data structures require safe memory reclamation that is delayed until all readers are finished with the data. Finally, atomic primitives can be a performance bottleneck themselves if they are used carelessly.

One example of a lock-free data structure is the Bw-Tree from Microsoft Research [29]. The high-level idea of the Bw-Tree is that it avoids locks by using an indirection layer that maps logical identifiers to physical pointers for the tree’s internal components. Threads then apply concurrent updates to a tree node by appending delta records to that node’s modification log. Subsequent operations on that node must replay these deltas to obtain its current state.
Storage Engine

Eliminate locks

B-Tree

Links

Payload

John  London  $80

Jane  Paris  $99

Larry  Rome  $70
Storage Engine

Links

Payload

John  London  $80
Jane  Paris  $99

Eliminate locks

Larry  Rome  $75
Larry  Rome  $70
Storage Engine

Which to read? It depends!
Hekaton

- Compiler
- Runtime
- Storage Engine

Eliminate instructions
Eliminate locks
Concurrency Control

Links

Payload

Which to read? It depends!

John London $80

Jane Paris $99

Larry Rome $75

Larry Rome $70
Concurrent Control

<table>
<thead>
<tr>
<th>Timestamps</th>
<th>Links</th>
<th>Payload</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>20</td>
<td>John</td>
</tr>
<tr>
<td>20</td>
<td>∞</td>
<td>Larry</td>
</tr>
<tr>
<td></td>
<td></td>
<td>London</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Jane</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Paris</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$80</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$99</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$75</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$70</td>
</tr>
</tbody>
</table>

B-Tree
Concurrent Control

<table>
<thead>
<tr>
<th>Timestamps</th>
<th>Links</th>
<th>Payload</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>20</td>
<td>Larry</td>
</tr>
<tr>
<td>Rome</td>
<td>$75</td>
<td></td>
</tr>
</tbody>
</table>

Determine **record visibility** by valid time (begin and end)

Assign transactions:
- Logical Read Time --- for visibility
- Commit time --- for serialization history
Concurrency Control

**Serializability** requires:

No updates to read records

Scans do not return new versions

**Validate** at commit time!

Authors claim this is cheap
High-Performance Concurrency Control
Mechanisms for Main-Memory Databases

Per-Åke Larson¹, Spyros Blanas², Cristian Diaconu¹,
Craig Freedman¹, Jignesh M. Patel², Mike Zwilling¹
Microsoft¹, University of Wisconsin – Madison²
{palarson, cdiaconu, craigfr,mikezw}@microsoft.com, {sblanas, jignesh}@cs.wisc.edu

ABSTRACT
A database system optimized for in-memory storage can support much higher transaction rates than current systems. However, standard concurrency control methods used today do not scale to the high transaction rates achievable by such systems. In this paper we introduce two efficient concurrency control methods specifically designed for main-memory databases. Both use multiversioning to isolate read-only transactions from updates but differ in how atomicity is ensured: one is optimistic and one is pessimistic. To avoid expensive context switching, transactions never block during normal processing but they may have to wait before commit to ensure correct serialization ordering. We also implemented a main-memory optimized version of single-version locking. Experimental results show that while single-version locking works well when transactions are short and contention is low performance degrades under more demanding conditions. The multiversion schemes have higher overhead but are much less sensitive to hotspots and the presence of long-running transactions.

found that traditional single-version locking is “fragile”. It works well when all transactions are short and there are no hotspots but performance degrades rapidly under high contention or when the workload includes even a single long transaction.

Decades of research has shown that multiversion concurrency control (MVCC) methods are more robust and perform well for a broad range of workloads. This led us to investigate how to construct MVCC mechanisms optimized for main memory settings. We designed two MVCC mechanisms: the first is optimistic and relies on validation, while the second one is pessimistic and relies on locking. The two schemes are mutually compatible in the sense that optimistic and pessimistic transactions can be mixed and access the same database concurrently. We systematically explored and evaluated these methods, providing an extensive experimental evaluation of the pros and cons of each approach. The experiments confirmed that MVCC methods are indeed more robust than single-version locking.

This paper makes three contributions. First, we propose an opti-
Other Details in Paper

• Commit dependencies

• Durability

• Garbage Collection
Hekaton

- Compiler
  - Eliminate instructions
- Runtime
  - Eliminate locks
- Storage Engine
Interpreters

SELECT * FROM T WHERE T.ID > @id

Query Plan

SCAN T

FILTER T.ID > @id
Interpreters

SELECT * FROM T WHERE T.ID > @id

filter::getNext( )
for ( ;; )
    row = child.getNext( )
    if ! filter( row )
        return row

Recursive calls

Easy to read

Query Execution
Hekaton Compiler

SELECT * FROM T WHERE T.ID > @id

label: filter_getNext
for ( ;; )
  goto scan_getNext
if ! filter( row )
  goto output

Minimize instructions

Hard to read

Query Execution
Hekaton Compiler

Payload

Larry   Rome   $75

Storage engine has no knowledge of records structures

Compile structures at table creation time
Other Details in Paper

• C vs. SQL type challenges

• Interoperability with SQL Server
Does it Work?

Hekaton compared to SQL Server:

- 10 – 20X reduction in CPU cycles
- 15X improvement in throughput
- Near linear scalability
Hekaton

Eliminates locks and instructions by Lock free data structures

Optimistic concurrency control

Compiled C code for stored procs

Completely within SQL Server!
Trekking Through Siberia: Managing Cold Data in a Memory-Optimized Database

Ahmed Eldawy*
University of Minnesota
dawy@cs.umn.edu

Justin Levandoski
Microsoft Research
justin.levandoski@microsoft.com

Per-Åke Larson
Microsoft Research
palarson@microsoft.com

ABSTRACT
Main memory databases can be the best solution for access patterns such as flash, but many record access rates are too high for economical data storage. Hekaton plans to separate hot and cold data by managing cold data on secondary storage while maximizing and minimizing memory usage. We accept an acceptable 7% drop in memory prices over the three-year period. Challenges include cold data classification and intra-data storage migration. How can we use the available cold data? Cold storage access reduction makes use of Bloom filters for point lookups and summaries of the cold store content. We are investigating two techniques: a version of Bloom filters for point lookups and a brief summary of related work. Columnstore indexes on in-memory tables are used; all we assume is that the cold store provides methods for maintaining a simple LRU chain added 25% overhead to the memory engine to speed up processing. This paper describes four such enhancements: column store indexes on in-memory tables, making secondary column store indexes on disk-based tables updatable, allowing B-tree indexes on primary column store indexes, and further speeding up the column store scan operator.

Real-Time Analytical Processing with SQL Server

Per-Åke Larson, Adrian Birka, Eric N. Hanson,
Weiyun Huang, Michal Nowakiewicz, Vassilis Papadimos
Microsoft
{palarson, adbirka, ehans, weiyh, michalno, vasilp}@microsoft.com

ABSTRACT
Over the last two releases SQL Server has integrated two specialized engines into the core system: the Apollo column store engine for analytical workloads and the Hekaton in-memory engine for high-performance OLTP workloads. There is an increasing demand for real-time analytics, that is, for running analytical queries and reporting on the same system as transaction processing so as to have access to the freshest data. SQL Server 2016 will include enhancements to column store indexes and in-memory tables that significantly improve performance on such hybrid workloads. This paper describes four such enhancements: column store indexes on in-memory tables, making secondary column store indexes on disk-based tables updatable, allowing B-tree indexes on primary column store indexes, and further speeding up the column store scan operator.

which is clearly prohibitively expensive. Vice versa, lookups are very fast in in-memory tables but complete table scans are expensive because of the large numbers of cache and TLB misses and the high instruction and cycle count associated with row-at-a-time processing.

This paper describes four enhancements in the SQL Server 2016 release that are designed to improve performance on analytical queries in general and on hybrid workloads, in particular.

1. **Columnstore indexes on in-memory tables.** Users will be able to create columnstore indexes on in-memory tables in the same way as they can now for disk-based tables. The goal is to greatly speed up queries that require complete table scans.

2. **Updatable secondary columnstore indexes.** Secondary CSIs on disk-based tables were introduced in SQL Server 2012. However, adding aCSI makes the table read-only. This limi.
Hekaton Discussion

Ruling out partitioning

Overhead of commit validation

Integration with SQL Server
(must explicitly declare table types)
Discard anything that does not bring you joy.