




Heckaton

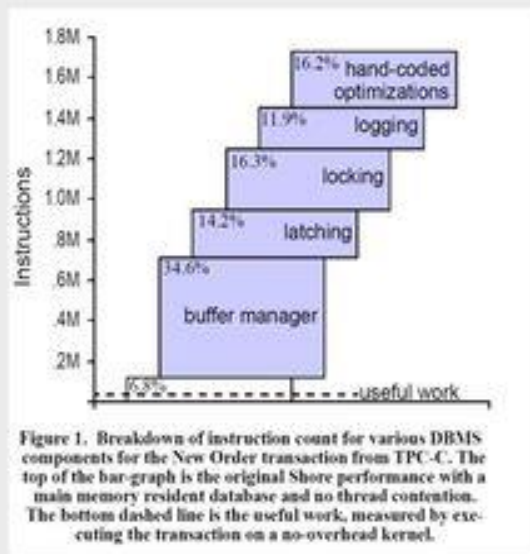
SQL Server's Memory Optimized OLTP Engine



Agenda

- Introduction to Hekaton
- Design Consideration
- High Level Architecture
- Storage and Indexing
- Query Processing
- Transaction Management
- Transaction Durability
- Garbage Collection
- Experimental Results

The Path to In-Memory OLTP



OLTP Through the Looking Glass, and What We Found There

by Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, Michael Stonebraker, SIGMOD 2008

Project Verde

- 2007

"Early"
Hekaton

- 2009/2010

Hekaton
becomes In-
Memory OLTP

- SQL Server 2014

In-Memory
OLTP
Unleashed

- SQL Server 2016

What is in Hekaton?

- An **engine** integrated into SQL Server.
- Provides **10x – 100 x improvement on performance critical queries**
- Uses **In memory optimized tables**
- Tables can be **queried and updated using T-SQL**



What is in Memory Optimized Table?

A memory-optimized table is one where SQL Server will always store **in memory** the **whole table and its indexes**.

Accessing in-memory data structures, user processes will *always* find the required data in-memory

Motivation: Increasing Query Throughput by 10x - 100x

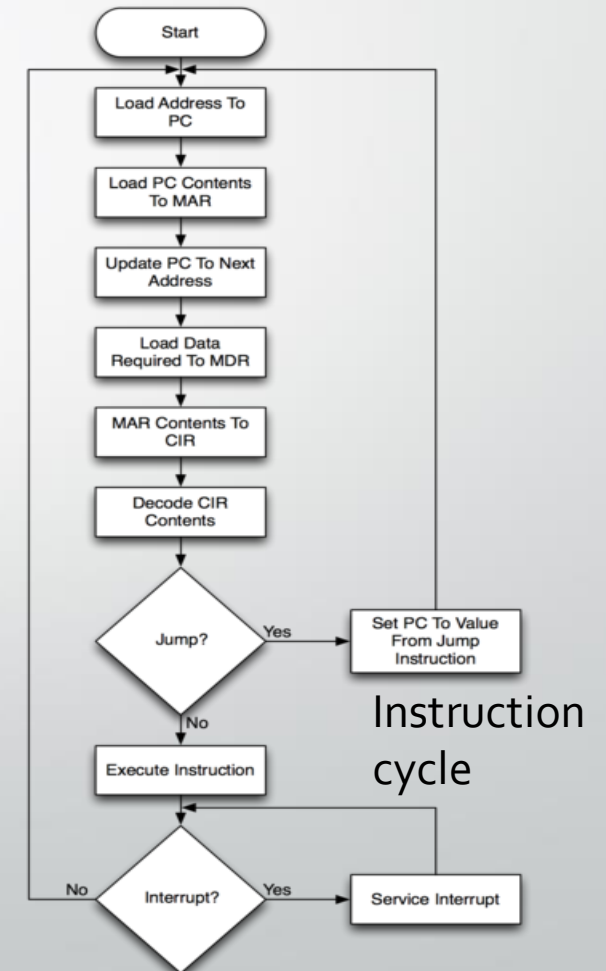
Query Throughput: Amount of transaction request processed / Time

How can we increase Query Throughput ?

- Improve scalability
- Reduce Cycles Per Instruction (CPI)
- Reduce number of instructions executed per transaction

After analysis

- Scalability and CPI improvement *does not produce significant results*
- **Reducing number of CPU instructions per transaction** *help us achieve 10x – 100x performance gain*
- In-memory tables can reduce CPU instructions significantly



Why Hekaton over other in memory RDBMS?

- Hekaton engine is integrated into SQL Server
No expense and hassle of another DBMS
- Performance critical table only reside in main memory
Efficient use of memory
- Stored Procedures accessing only Hekaton tables can be compiled into native machine code
Performance gain!
- Conversion can be done one table / stored procedure at a time
Easier Integration!

Design Considerations

Reduce Number of Instructions Per Transaction Request

- **Architectural Principles**
 - Optimized Indexes for Main Memory
 - Eliminate latches and locks
 - Compile request to native code
- **No Partitioning**

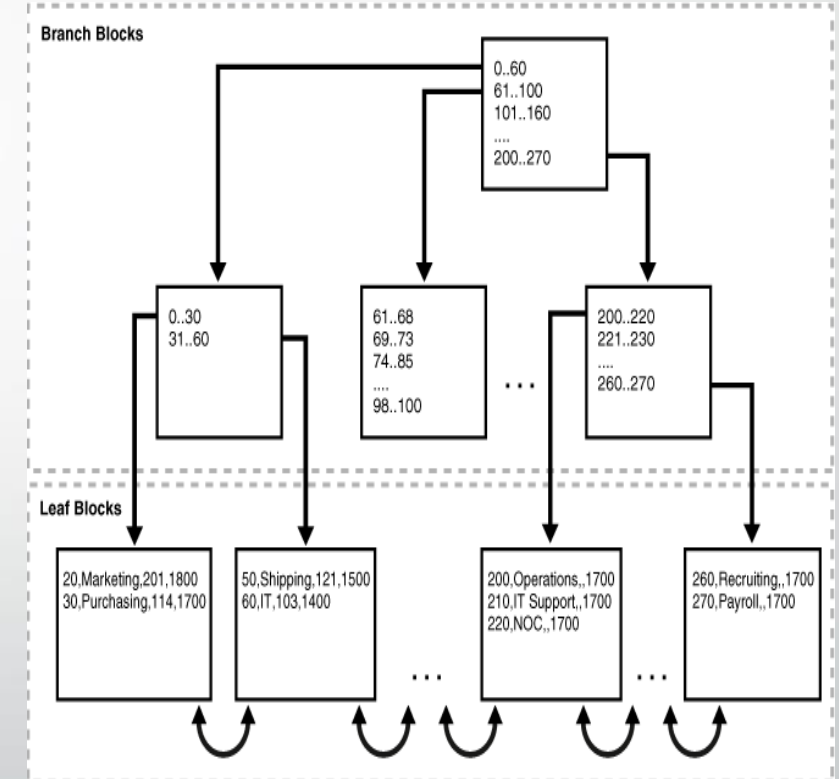
Optimizing Indexes for main memory

SQL Server Engine

- **Indexes are managed by B-Trees**, used to get records stored in disk pages.
- Key lookup in B-Tree required thousands of instructions.
- Disk reads for reading both index pages and record pages.
- Required a buffer pool, page is latched before being accessed

Hekaton Engine

- Uses **hash indexes** and **range indexes**
- Designed and optimized for memory resident data



Compile Request to Native Code

SQL Server Engine

- **Interpreter based execution**, a simple transaction with few lookups require thousands of instructions.
- Required a buffer pool, page is latched before being accessed

Hekaton Engine

- Maximizes runtime performance by **compiling stored procedures in to machine code**.
- No runtime overheads

Eliminates Latches and Locks

SQL Server Engine

- Shared memory decreases scalability
- Has to maintain data structures for **latches, spin locks and lock manager**
- Latches and Locks requires shared memory between processes.

Hekaton Engine

- All data structures are entirely lock free
- Optimistic concurrency control, multi-versioning and latch

No Partitioning in Hekaton Engine

Any thread can access any part of database

Other Databases

- Partition database by core
- Give one core exclusive access to a partition

Why No Partitioning?

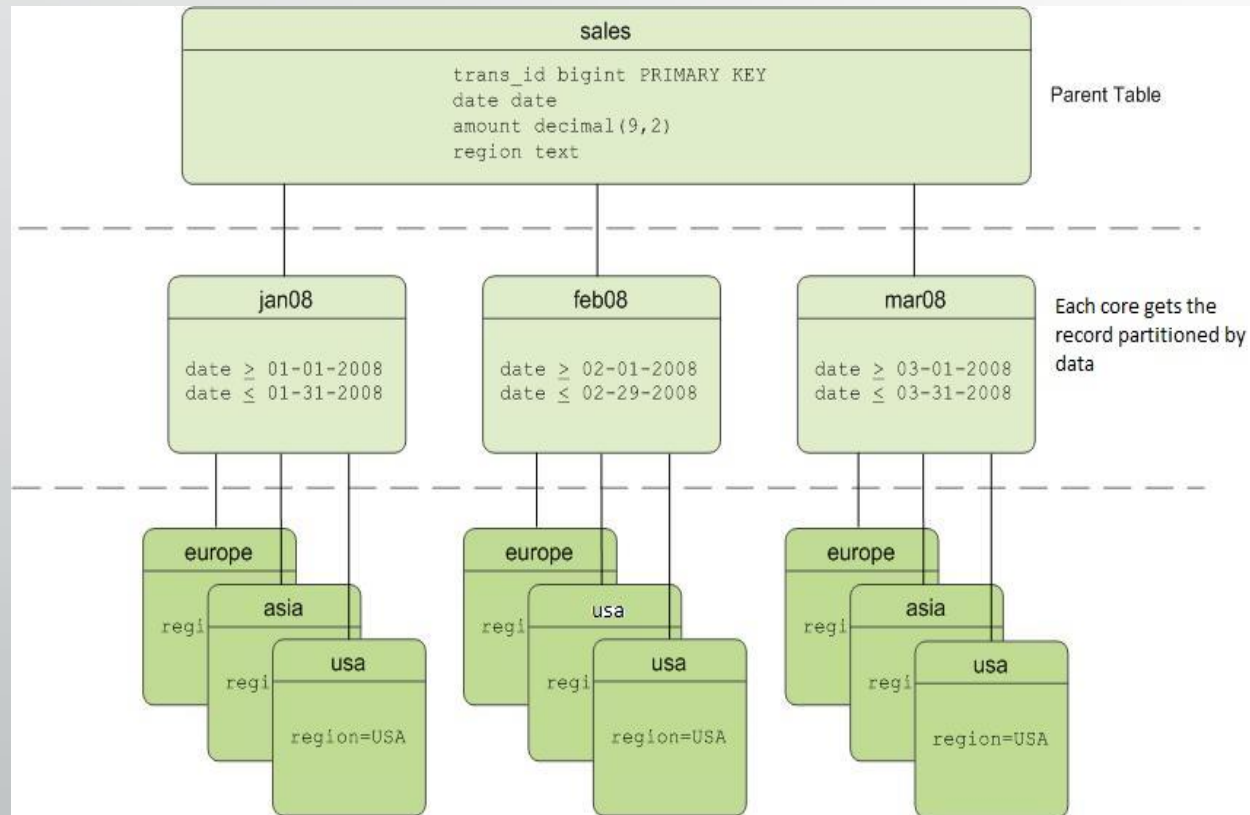
- Overhead of **constructing, sending and receiving request** across the cores
- Overhead of **returning result** from each core

In a non-partitioned system a *single lookup* is done in a *hash table*

Overhead of Partitioning

Works great only if work load is partitionable

Suppose we have a sales table partitioned by column A across the cores



Query 1

Select * from sales where date = 03-01-2008

In this case, we just need to access node 3.

Query 2

Select * from sales where region = 'asia'

In this case, all cores gets accessed

Core 2 gets accessed even though there is no record found there.

Each request has to be dequeue by receiving thread, processed and results returned!

High Level Architecture

Three major component of Hekaton



Storage Engine

- Transaction Operations
- Apply hash and range indexes on tables



Compiler

- Compile Procedures in native highly efficient machine code.



Runtime System

- Integrates SQL Server Resources
- Serves as common library of additional functionality needed by compiled stored procedures

SQL Server Components Used By Hekaton



Storage Engine



Compiler



Runtime System

Metadata

- Stores metadata about tables and index

Query Optimization

- Query during compilation are optimized using regular SQL optimizer

Query Interop

- To provide operators for inserting, deleting and updating data by SQL Server stored procedures

Transactions

- To Enable SQL Server transactions update both regular and Hekaton tables

High Availability

- Integrated with AlwaysOn, uses SQL Server High Availability feature.

Storage, Log

- Stores checkpoints for recovery of indexes and tables on recovery
- Serves as common library of additional functionality needed by compiled stored procedures

Storage and Indexing

1. Support 2 types of indexes

- **Hash Indexes** – *for equality search*
Implemented using lock free hash tables
- **Range Indexes** – *for range and equality search*
Implemented using Bw-trees (lock free version of B-trees)

2. A table can have multiple indexes

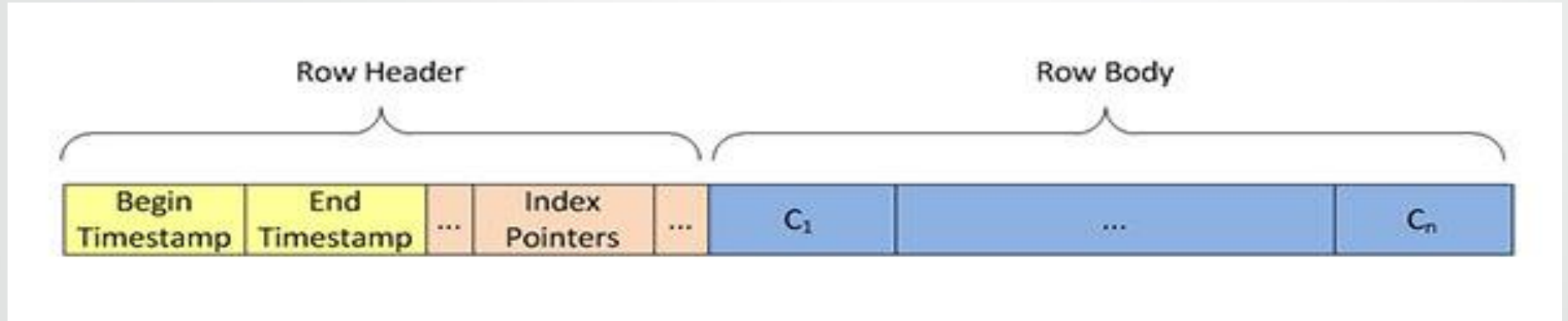
3. Records are always accessed via index lookup

4. Uses multi-versioning – each updates creates a new version of the record in Hekaton table

5. Indexes do not exist on disk – recreated during recovery

Hekaton Table Record Format

Maintains multiple version of same record



Index Pointers

- A pointer for hash index
- A pointer for range index

Timestamp

- Begin/End timestamp determines row's validity
- Different version of record have non overlapping valid time

Key Notes

- There is no data or index page just rows
- This structure of row optimizes in-memory residency and access

SQL Server Index vs Hekaton

Suppose we are running a query with equality predicate on a non-primary key

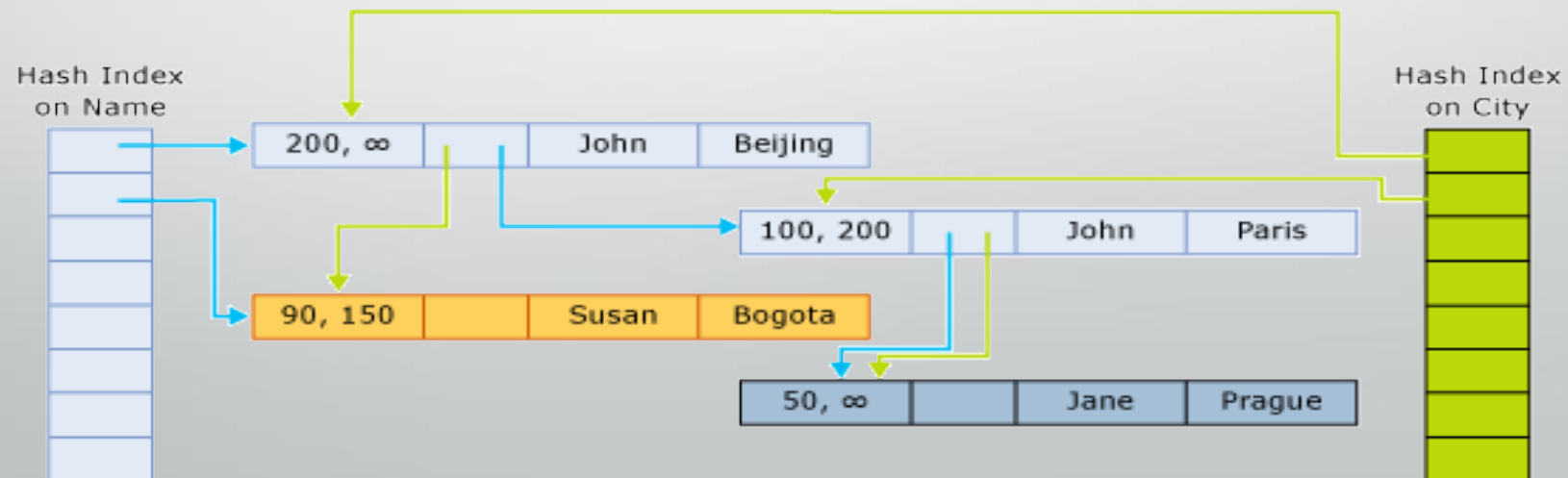


In SQL Server B-tree

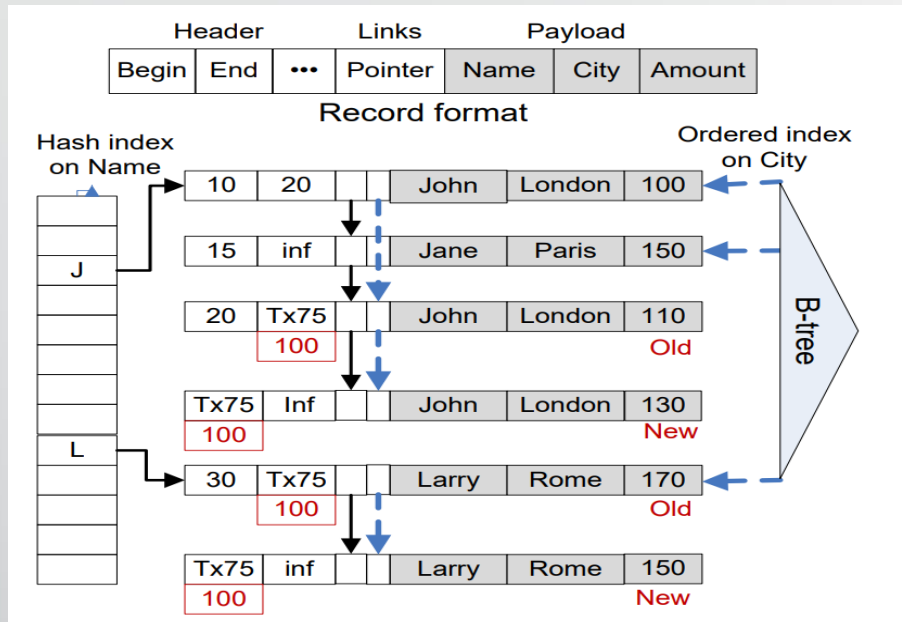
1. We pick the root page of non-clustered index.
2. We traverse through index pages by reading index rows.
3. Once we go to the leaf node we get the record, if a query is not satisfied by record we use clustered index

So many page read!

You can have multiple hash indexes in Hekaton



Hekaton Table



Read Operation

- Specifies a logical read time
- Only version whose valid time overlaps the read time are visible

Example: name = john and valid time 10 to 20

Hash Indexes

Create buckets based on Key

Range Index (Bw-tree)

Leaf nodes stores pointers to records

Key Notes:

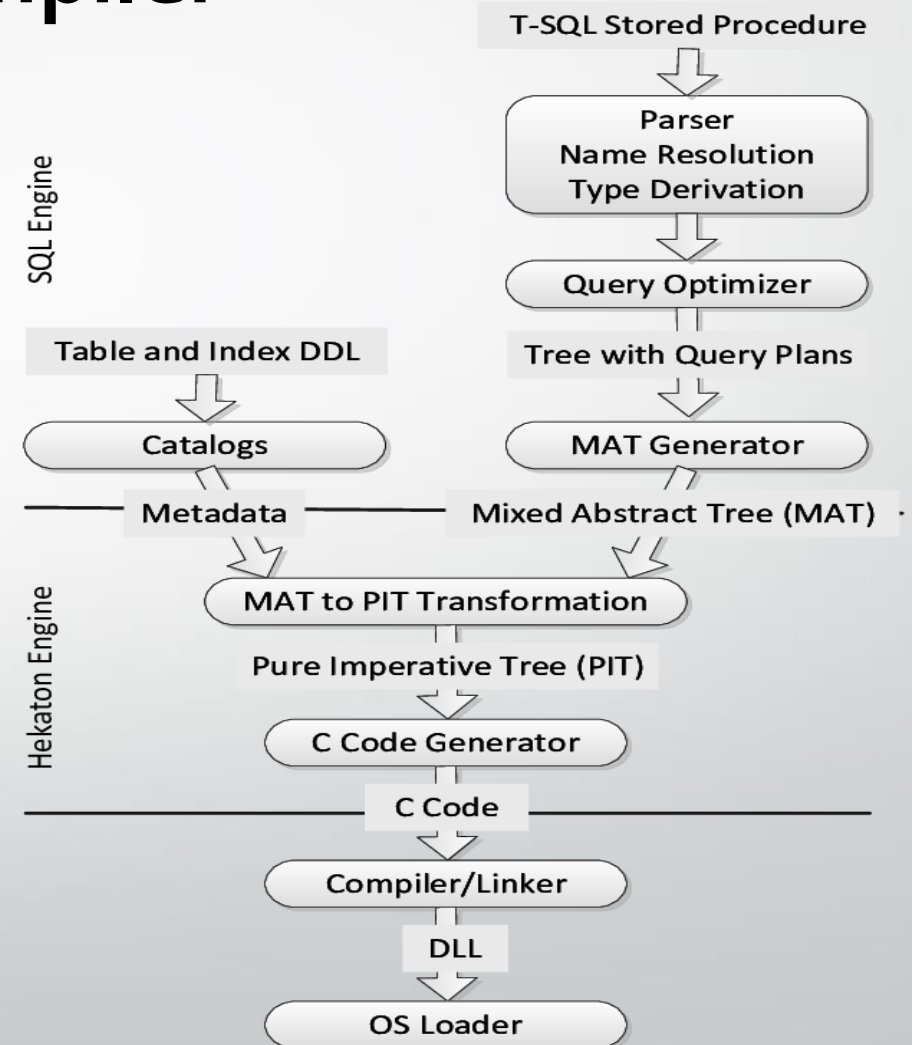
- A bucket might have multiple version of records
- End Timestamp with infinity shows latest version of records

Update Operation (Delete is similar)

- An update creates a new version
- Before commit stores the transaction id in begin timestamp of new version and end timestamp of old version, **prevents other transaction from updating the version**
- After commit update the replaces updated timestamp with transaction id

Architecture of Hekaton Compiler

- Reuses much of SQL Server T-SQL compilation stack
- Compiler is invoked only during **creation of memory optimized table** and **compilation of stored procedures**
- SQL Server compilation stack **returns a data structure called Mixed Abstract Tree (MAT)**
- MAT is **capable of representing metadata, imperative login expression and query plan**
- MAT is transformed to **Pure Imperative Tree (PIT)**
- PIT is simpler data structure that can easily be converted to C code.





Transaction Management

Transaction Management

Hekaton utilizes **Multiversion Concurrency Control (MVCC)**

A transaction is **serializable** if it reads and writes logically occur at the same time, that's is all it reads would see exactly the same data at the end of transaction.

Remember read is based on time!

A serializable transaction must hold the two properties

1. **Read Stability** – All version visible before transaction are also visible after transaction
2. **Phantom Avoidance** – Transaction scan **should not add addition new versions**

Time Stamp and Version Visibility

- **Logical Read Time of Transaction**

- * Any value between transaction begin time and current time
- * Version whose valid time overlaps logical read time are visible to transaction

- **Commit / EndTime**

- * Time when transaction modifies data commits
- * Determines transaction's position in serialization history

- **Valid Time of Versions**

- * Version begin time denotes the commit time of transaction that created the version
- * Version end time denotes the commit time of transaction that deletes the version
- * Valid time is the time range (begin time – end time) in which the version is visible to transaction.

Transaction Commit Processing - Validation

- For validation we need to verify **read stability** and **phantom** avoidance to ensure the transaction was serializable
- The transaction maintains the following
 1. **Read Set** – list of pointers to the version it has read (**Check Read Stability**)
 2. **Scan Set** – information needed to repeat scans (**Check Phantom Avoidance**)

Transaction Commit Processing - Dependencies

- A transaction T_1 that begins while transaction T_2 is in validation phase becomes dependant on T_2 .
- In this case T_1 has **commit dependency** on T_2 – that is T_1 is only allowed to commit if T_2 commits, **even though it can make changes to versions**.

Raises two problems

1. Transaction cannot commit until all transaction it is dependent on has committed.

Solution: Maintaining dependency counter for each transaction dependant on. When T_2 commits it decrements dependency counter maintained by T_1

2. This imply working with uncommitted data which should not be exposed to user

Solution: Read Barriers introduced that is transaction result is held back and not revealed to client until commit dependencies have resolved

Commit Logging and Post Processing

- Transaction T writes to transaction log the content of all new version created by T and primary key of all new version deleted.
- Transaction T is irreversibly committed after creation of transaction logs
- A post processing process starts in which begin and end timestamp of all version are updated to end timestamp of transaction

Transaction Rollback

- Occur due to user request or due to failures in commit processing
- Rollback is achieved by
 1. Invalidating all versions created by transaction
 2. Clearing the end timestamp of all field deleted by transaction
- Other transactions dependent on outcome if rolled-back transactions are notified



Transaction Durability

Transaction Durability

Recover Memory Optimized table after failure

Achieves using **Transaction Logs** and **Checkpoints** to durable storage

Design of Logging, Checkpointing and recovery guided by following principles

- Optimize Sequential Access
More money spend on main memory
- Push work to recovery time
Minimise overhead during normal transaction execution
- Eliminate scaling bottlenecks
- Enable parallelism in I/O and CPU during recovery

Data Stored on External Storage

Log Streams – Transaction Logging

- Contains effect of committed transactions – insertion and deletion of row versions
- Log records is generated at Transaction commit time
- Stored in SQL Server Transaction Logs

Checkpoint Stream - Checkpoints

- Consist of two types **data streams** and **delta streams**
- **Data Streams:** Contain all inserted versions during a timestamp interval
- **Delta Streams:** Associated with a particular data stream, contains dense list of integers identifying deleted versions.
- Stored in SQL Server File Streams
- Compressed representation of Log Streams
- Allow log streams to be truncated and improve crash recovery performance

Transaction Recovery

SQL Server scan tail of Hekaton transaction log to find most recent checkpoint



SQL Server communicates location of checkpoint inventory to Hekaton Engine



SQL Server and Hekaton recovery process start in parallel.



Checkpoint delta file used to filter data files not to be loaded. **No need to load deleted versions**



Checkpoint data file load proceeds in parallel across multiple file stream at file pair granularity



Tail of transaction log is replayed from the timestamp of the checkpoint



Garbage Collection

Garbage Collection (GC)

Cleanup versions that are no longer visible to active transactions

Desirable Properties for Garbage Collection(GC)

- **Hekaton GC in non-block**
Should never block processing of any active transaction
- **GC subsystem is cooperative**
Remove Garbage on encounter, be proactive
- **Processing is incremental**
Can be stopped and started anytime easily to avoid consuming CPU resources
- **GC is parallelized and scalable**
Multiple thread works on various phases with no cross-thread synchronization

GC Correctness

- Identify Garbage Versions

A version is garbage if

- Updated/deleted by committed transaction
- Can not be read by any transaction
- Created by transaction that rolls back

- Periodically scans global transaction map to determine oldest active transaction

Any version whose **end timestamp is less than current oldest active transaction** is not visible to any transaction

.

GC Removal

- Unlink a version from all indexes in which it participates
- Version for Removal collected in two ways
 1. Cooperative mechanism used by thread running transaction workload
Index Scanner encounter garbage as they scan and are empowered to unlink garbage version
 2. Parallel background collection process
Remove versions with no links or unlink version itself

GC Scalability

- A single GC process prodigally calculate current oldest active transaction, after this transaction are read for collection
- Multiple threads collect garbage versions in isolation
- Provides two scalability benefits
 1. Parallelizes works across CPU cores
 2. Allows GC system to self throttle



Experimental Results

Used workstation with 2.67 GHz and 8 GB Memory

CPU Efficiency - Lookups(Read)

- Random lookups in a table with 10M rows
- All data in memory
- 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 performance: 2.7M

- Speedup is 20x when performing 10 or more lookups per call
- 5% of CPU cycles used by the regular SQL Server
- 2.7M lookups per second per core

CPU Efficiency - Updates

<ul style="list-style-type: none">• Random updates, 10M rows, one index, snapshot isolation• Log IO disabled (disk became bottleneck)• Intel Xeon W2520 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
<ul style="list-style-type: none">• Hekaton performance: 1.9M updates/sec/core			

- Speedup is 30x when performing 100 or more update transactions
- 3-5% of CPU cycles used by the regular SQL Server Engine
- 1.9M updates per second per core



Log Data Performance

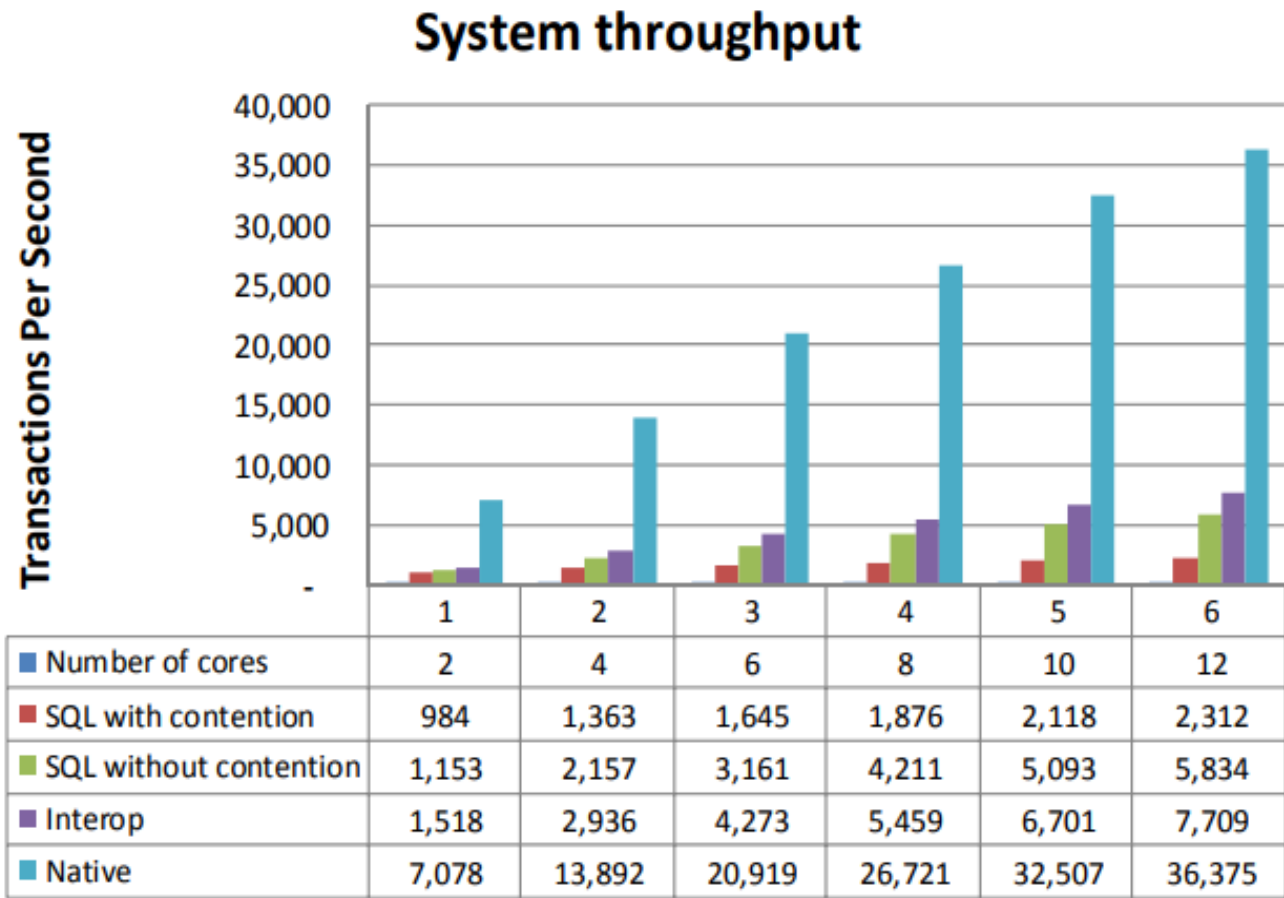
Log less data – **Reduced log output by 57 %**

Scaling Under Contention

Optimizing SQL Server Engine with motivation from Hekaton

- Scaling mean improving performance with increasing number of cores
- Hekaton is designed to eliminate lock and latch contention, which allows it to scale.
- What if we try to optimize SQL server, where root cause of scalability limitations is contentions?

Scaling Under Contention



SQL with contention – SQL Server Stored Procedures where transactions do not interfere to add contentions

SQL without contention – SQL server Stored Procedures that do not interfere no contentions

Interop – SQL Server stored procedures using Hekaton in-memory tables

Native – Accessing table through compiled stored procedures in SQL Server



Questions