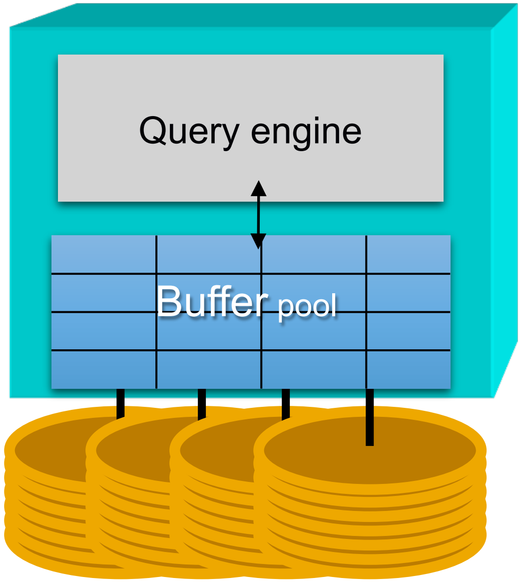
Evolving the Architecture of Sql Server

Paul Larson, Microsoft Research

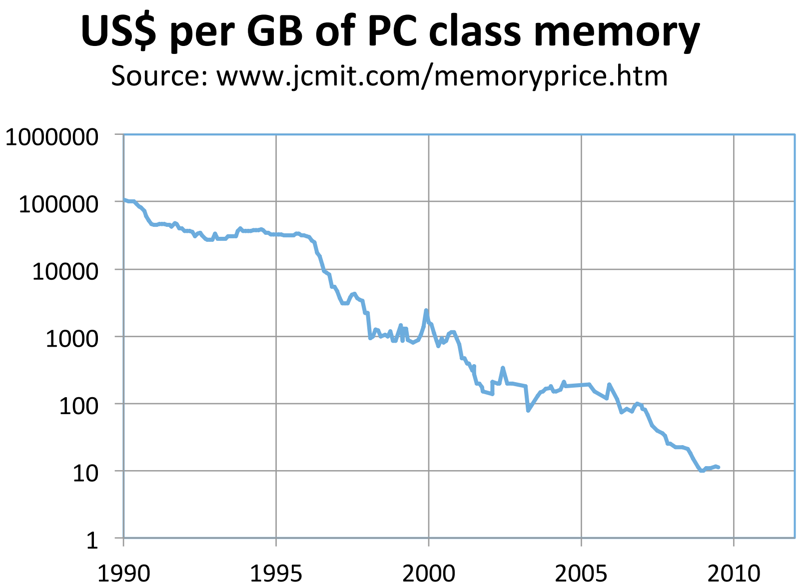
Microsoft® Research

Time travel back to circa 1980

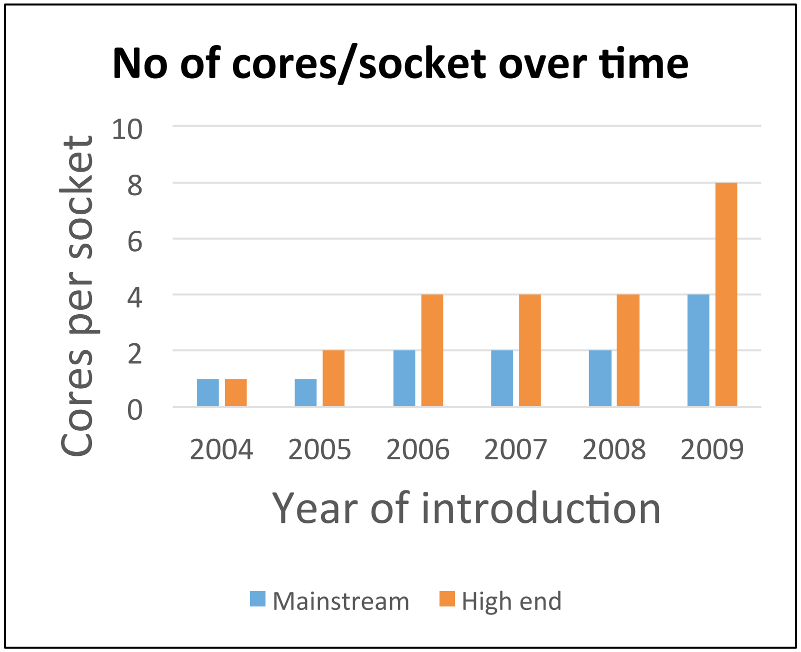
- Typical machine was VAX 11/780
  - 1 MIPS CPU with 1KB of cache memory
  - 8 MB memory (maximum)
  - 80 MB disk drives, 1 MB/second transfer rate
  - $250K purchase price!
- Basic DBMS architecture established
  - Rows, pages, B-trees, buffer pools, lock manager, ….
- Still using the same basic architecture!



But hardware has evolved dramatically

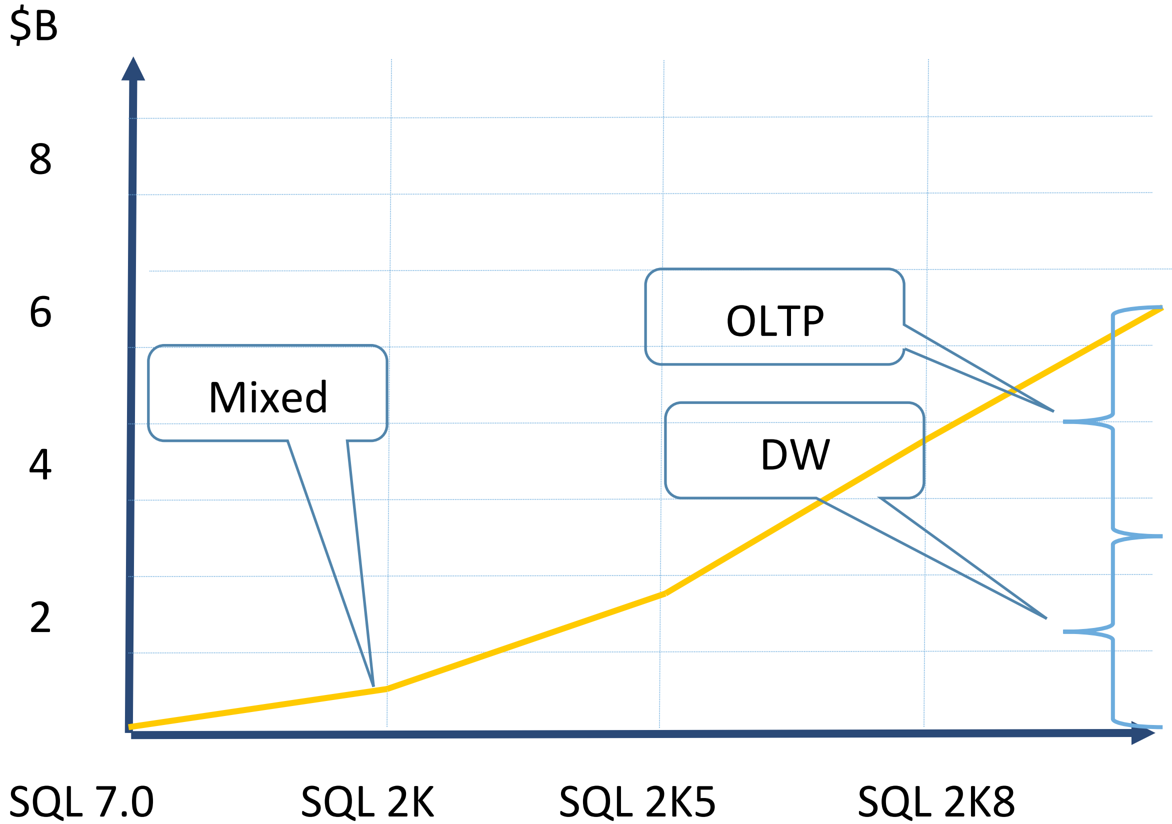


Shrinking memory prices

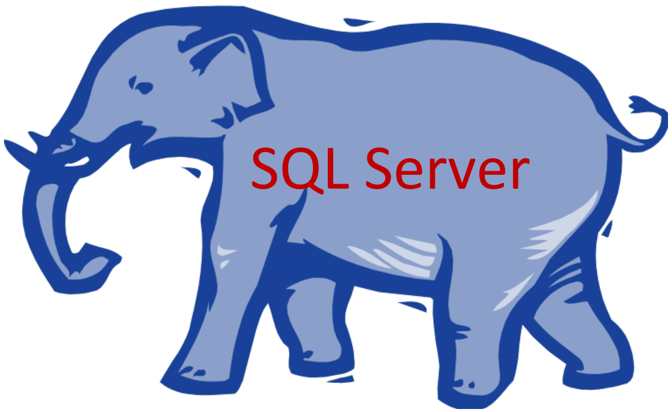


Stalling clock rates but more and more cores…

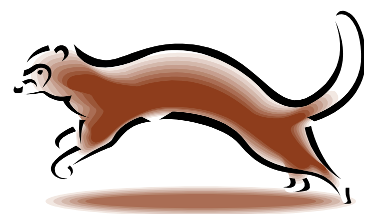
Workloads evolve too…



Are elephants doomed?

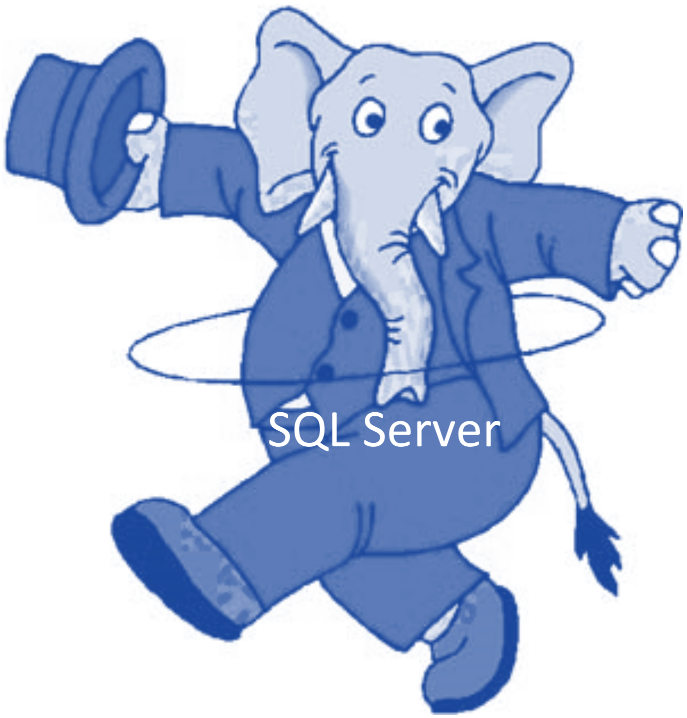


Main-memory DBMSs



Column stores

Make the elephant dance!



Hekaton

Apollo

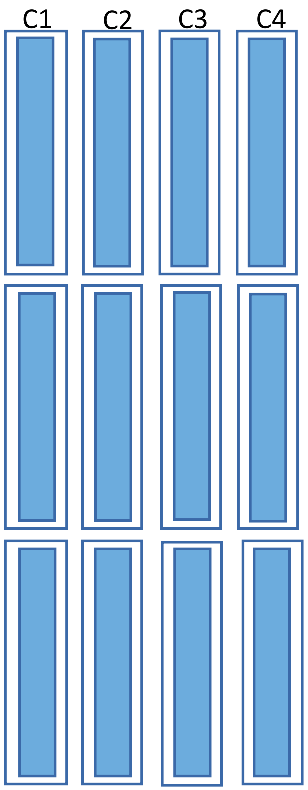
OK, time to get serious…

- Apollo
  - Column store technology integrated into SQL Server
  - Targeted for data warehousing workloads
  - First installment in SQL 2012, second in SQL 2014
- Hekaton
  - Main-memory database engine integrated into SQL Server
  - Targeted for OLTP workloads
  - Initial version in SQL 2014

- This talk doesn't cover
  - PDW – SQL Server Parallel Data Warehouse appliance
  - SQL Azure – SQL Server in the cloud

What is a column store index?

A B-tree index stores data row-wise



A column store index stores data column-wise

- Each page stores data from a single column
- Data <u>not</u> stored in sorted order
- Optimized for scans

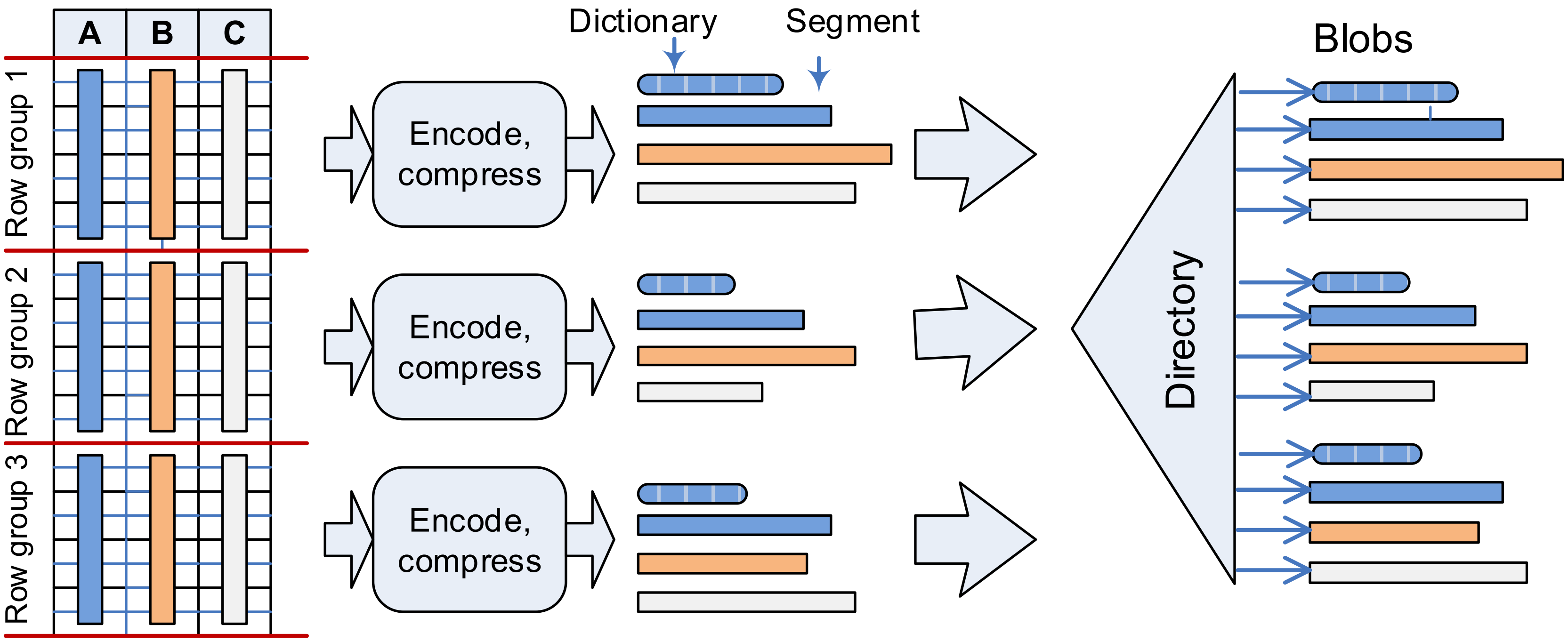
Project Apollo challenge

- Column stores beat the pants off row stores on DW workloads
  - Less disc space due to compression
  - Less I/O – read only required columns
  - Improved cache utilization
  - More efficient vector-wise processing
- Column store technology per se was not the problem
  - Old, well understood technology
  - Already had a fast in-memory column store (Analysis Services)
- Challenge: How to integrate column store technology into SQL Server
  - No changes in customer applications
  - Work with <u>all</u> SQL Server features
  - Reasonable cost of implementation

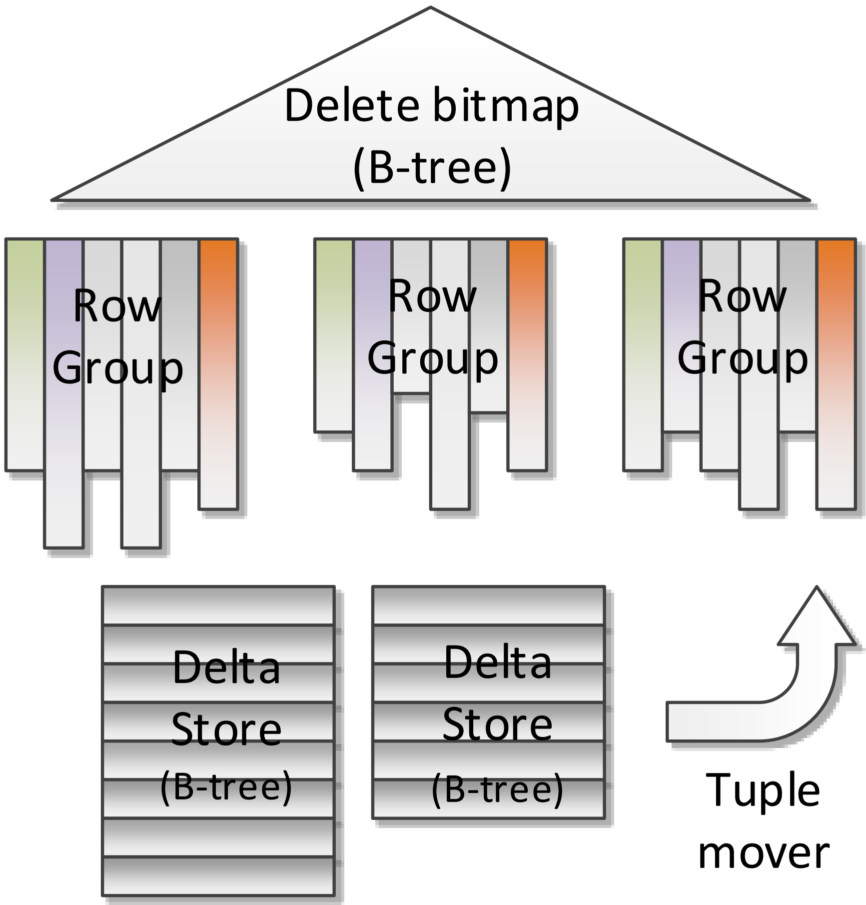
Key design decisions

- Expose column stores as a new index type
  - One new keyword in index create statement (COLUMNSTORE)
  - No application changes needed!
- Reuse existing mechanisms to reduce implementation cost
  - Use Vertipaq column store format and compression
  - Use regular SQL Server storage mechanisms
  - Use a regular row store for updates and trickle inserts
- Add a new processing mode: batch mode
  - Pass large batches of rows between operators
  - Store batches column-wise
  - Add new operators that process data column-wise

Creating and storing a column store index



Update mechanisms



- Delete bitmap
  - B-tree on disk
  - Bitmap in memory
- Delta stores
  - Up to 1M rows/store
  - Created as needed
- Tuple mover
  - Delta store → row group
  - Automatically or on demand

So does it pay off?

- Index compression ratio highly data dependent
  - Regular: 2.2X – 23X; archival: 3.6X – 70X
- Fast bulk load: 600GB/hour on 16 core system
- Trickle load rates (single threaded)
  - Single row/transaction: 2,944 rows/sec
  - 1000 rows/transaction: 34,129 rows/sec

Customer experiences (SQL 2012)

- Bwin
  - Time to prepare 50 reports reduced by 92%, 12X
  - One report went from 17 min to 3 sec, 340X
- MS People
  - Average query time dropped from 220 sec to 66 sec, 3.3X
- Belgacom
  - Average query time on 30 queries dropped 3.8X, best was 392X

Where do performance gains come from?

- Reduced I/O
  - Read only required columns
  - Better compression
- Improved memory utilization
  - Only frequently used columns stay in memory
  - Compression of column segments
- Batch mode processing
  - Far fewer calls between operators
  - Better processor cache utilization – fewer memory accesses
  - Sequential memory scans
  - Fewer instructions per row

Current status

- SQL Server 2012
  - Secondary index only, not updateable
- SQL Server 2014
  - Updateable column store index
  - Can be used as base storage (clustered index)
  - Archival compression
  - Enhancements to batch mode processing

Hekaton: what and why

- Hekaton is a high performance, memory-optimized OLTP engine integrated into SQL Server and architected for modern hardware trends

- Market need for ever higher throughput and lower latency OLTP at a lower cost
- HW trends demand architectural changes in RDBMS to meet those demands

Hekaton Architectural Pillars

Main-Memory Optimized

- Optimized for in-memory data
- Indexes (hash, range) exist only in memory
- No buffer pool
- Stream-based storage (log and checkpoints)

Designed for High Concurrency

- Multi-version optimistic concurrency control with full ACID support
- Core engine using lock-free algorithms
- No lock manager, latches or spinlocks

T-SQL Compiled to Machine Code

- T-SQL compiled to machine code via C code generator and VC
- Invoking a procedure is just a DLL entry-point
- Aggressive optimizations @ compile time

Integrated into SQL Server

- Integrated queries & transactions
- Integrated HA and backup/restore
- Familiar manageability and development experience

Steadily declining memory price	Many-core processors	Stalling CPU clock rate	TCO
Hardware trends			Business Driver

Hekaton does not use partitioning

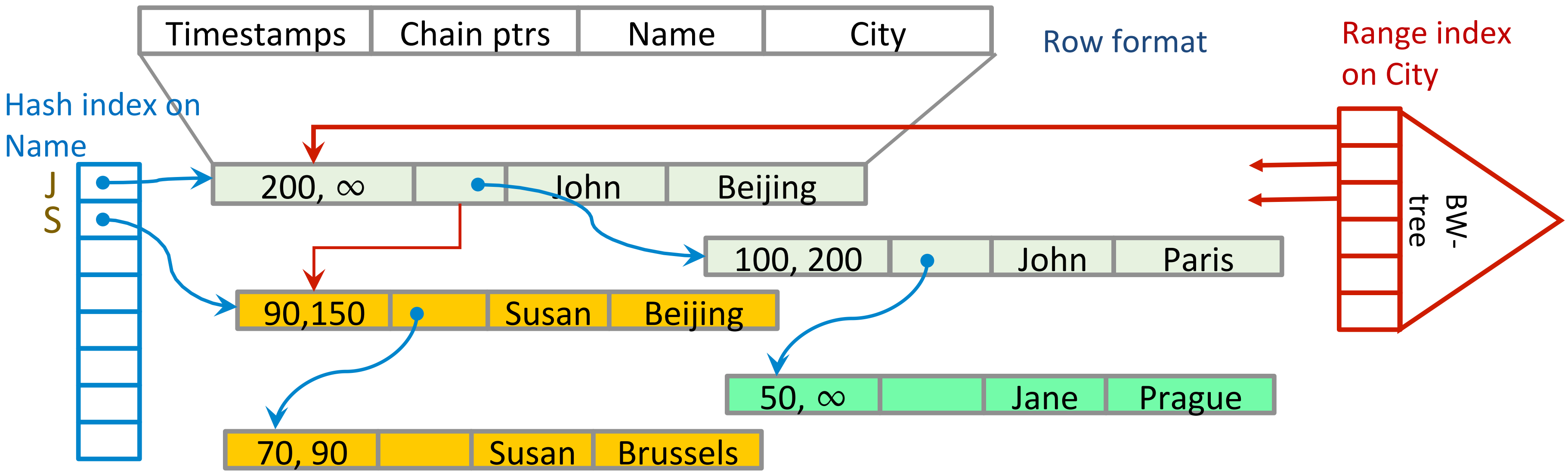
- Partitioning is a popular design choice
  - Partition database by core
  - Run transactions serially within each partition
  - Cross-partition transactions problematic and add overhead
- Partitioning causes unpredictable performance
  - Great performance with few or no cross–partition transactions
  - Performance falls off a cliff as cross-partition transactions increase
- But many workloads are <u>not</u> partitionable
- SQL Server used for many different workloads
  - Can’t ship a solution with unpredictable performance

Data structures for high concurrency

1. Avoid global shared data structures
   - Frequently become bottlenecks
   - Example, no lock manager
2. Avoid serial execution like the plague
   - Amdahl's law strikes hard on machines with 100's of cores
3. Avoid creating write-hot data
   - Hot spots increase cache coherence traffic

- Hekaton uses only latch-free (lock-free) data structures
  - Indexes, transaction map, memory allocator, garbage collector, ….
  - No latches, spin locks, or critical sections in sight
- One single serialization point: get transaction commit timestamp
  - One instruction long (Compare and swap)

Storage optimized for main memory



- Rows are multi-versioned
- Each row version has a valid time range indicated by two timestamps
- A version is visible if transaction read time falls within version's valid time
- A table can have multiple indexes

What concurrency control scheme?

- Main target is high-performance OLTP workloads
  - Mostly short transactions
  - More reads than writes
  - Some long running read-only queries

- Multiversioning
  - Pro: readers do not interfere with updaters
  - Con: more work to create and clean out versions
- Optimistic
  - Pro: no overhead for locking, no waiting on locks
  - Pro: highly parallelizable
  - Con: overhead for validation
  - Con: more frequent aborts than for locking

Hekaton transaction phases

Txn events	**Txn phases**	
Begin		• Get txn start timestamp, set state to Active
	Normal processing	• Perform normal processing • remember read set, scan set, and write set
Precommit		• Get txn end timestamp, set state to Validating
	Validation	• Validate reads and scans • If validation OK, write new versions to redo log
Commit		• Set txn state to Committed
	Post-processing	• Fix up version timestamps • Begin TS in new versions, end TS in old versions
Terminate		• Set txn state to Terminated
		• Remove from transaction map

Transaction validation

- **Read stability**
  - Check that each version read is still visible as of the end of the transaction
- **Phantom avoidance**
  - Repeat each scan checking whether new versions have become visible since the transaction began

- Extent of validation depends on isolation level
  - Snapshot isolation: no validation required
  - Repeatable read: read stability
  - Serializable: read stability, phantom avoidance

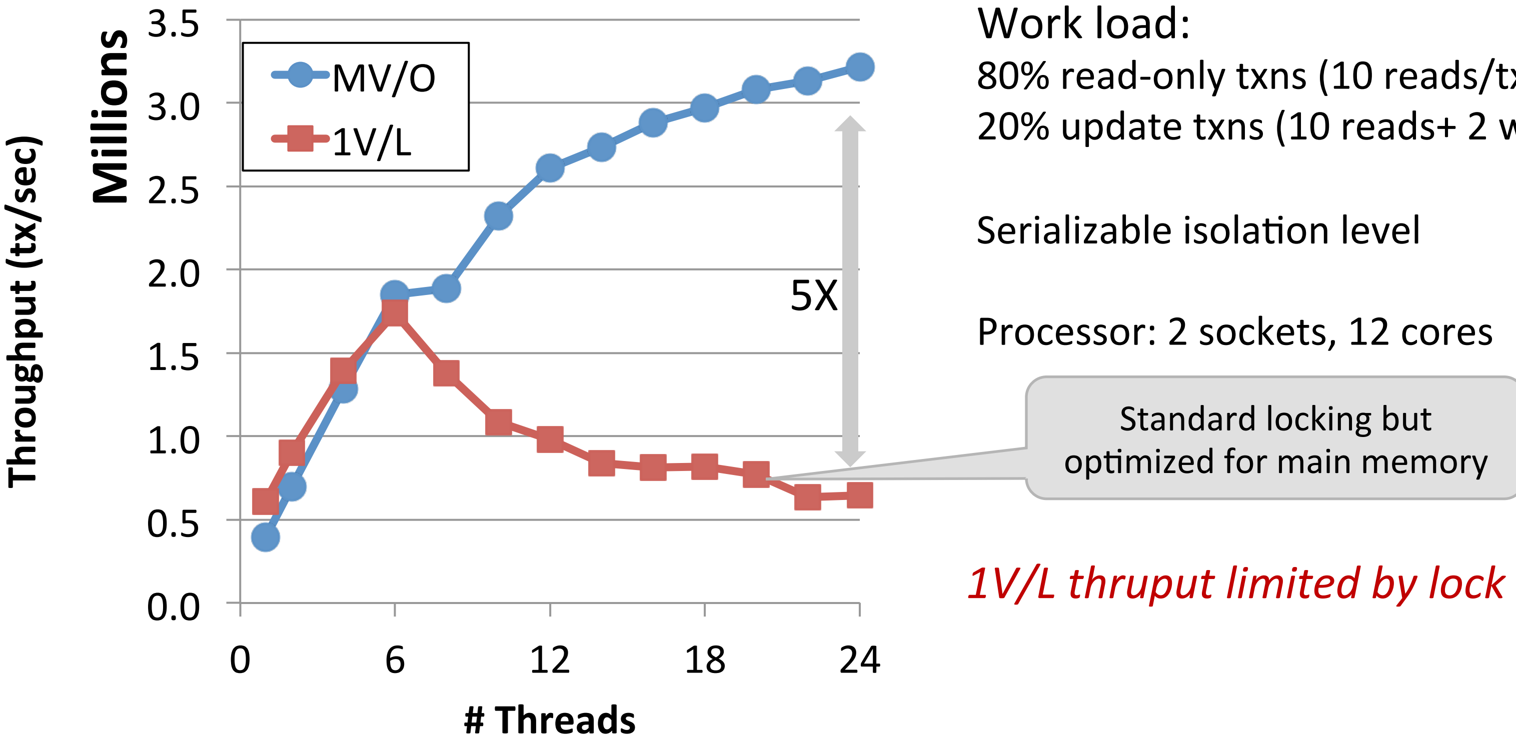
Details in "High-Performance concurrency control mechanisms for main-memory databases", VLDB 2011

Non-blocking execution

- Goal: enable highly concurrent execution
  - no thread switching, waiting, or spinning during execution of a transaction
- Lead to three design choices
  - Use only latch-free data structure
  - Multi-version optimistic concurrency control
  - Allow certain speculative reads (with commit dependencies)
- Result: great majority of transactions run up to final log write without ever blocking or waiting

- What else may force a transaction to wait?
  - Outstanding commit dependencies before returning a result to the user (rare)

Scalability under extreme contention
(1000 row table, core Hekaton engine only)



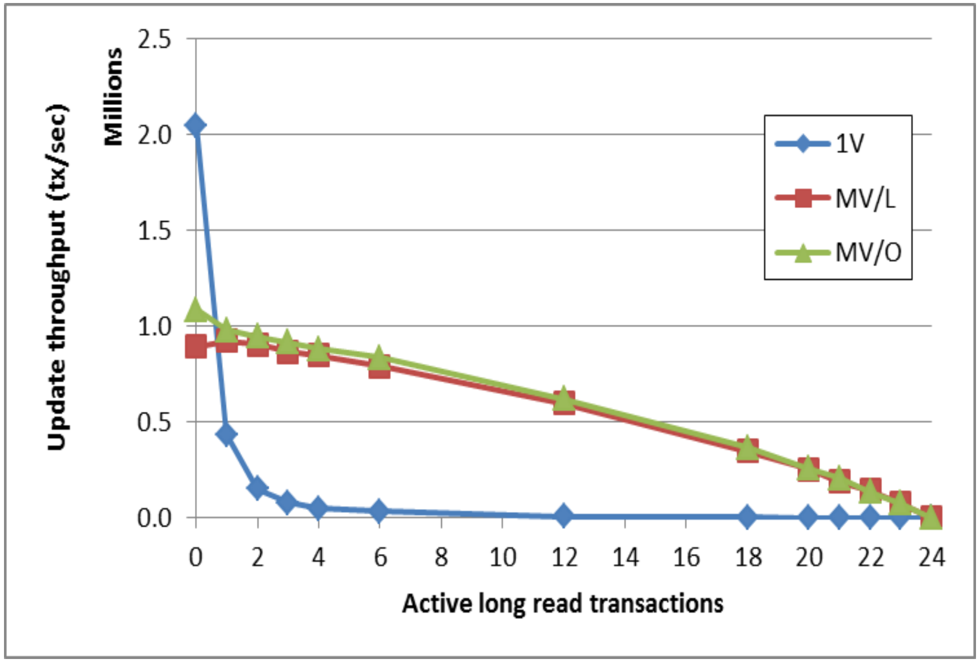
Work load:
80% read-only txns (10 reads/txn)
20% update txns (10 reads+ 2 writes/txn)

Serializable isolation level

Processor: 2 sockets, 12 cores

1V/L thruput limited by lock thrashing

Effect of long read-only transactions



Workload:

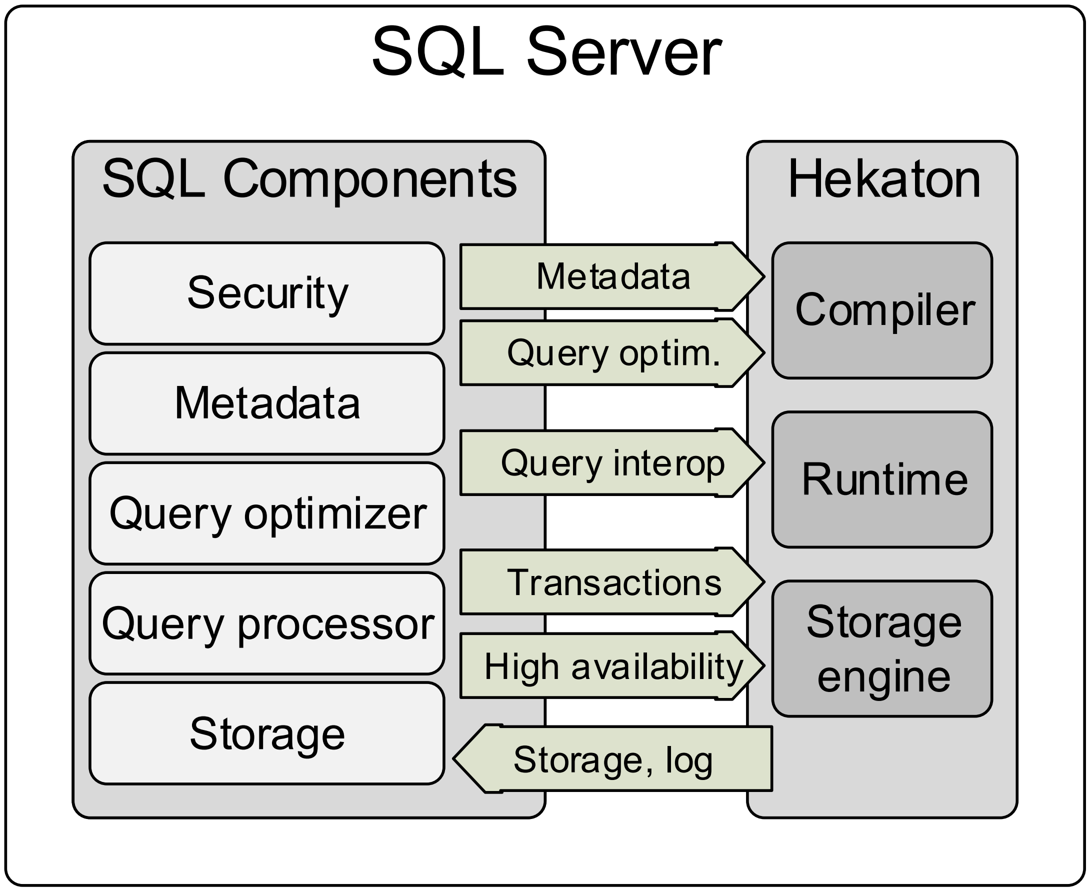
- Short txns 10R+ 2W
- Long txns: R 10% of rows

24 threads in total

- X threads running short txns
- 24-X threads running long txns

- Traditional locking: update performance collapses
- Multiversioning: update performance per thread unaffected

Hekaton Components and SQL Integration

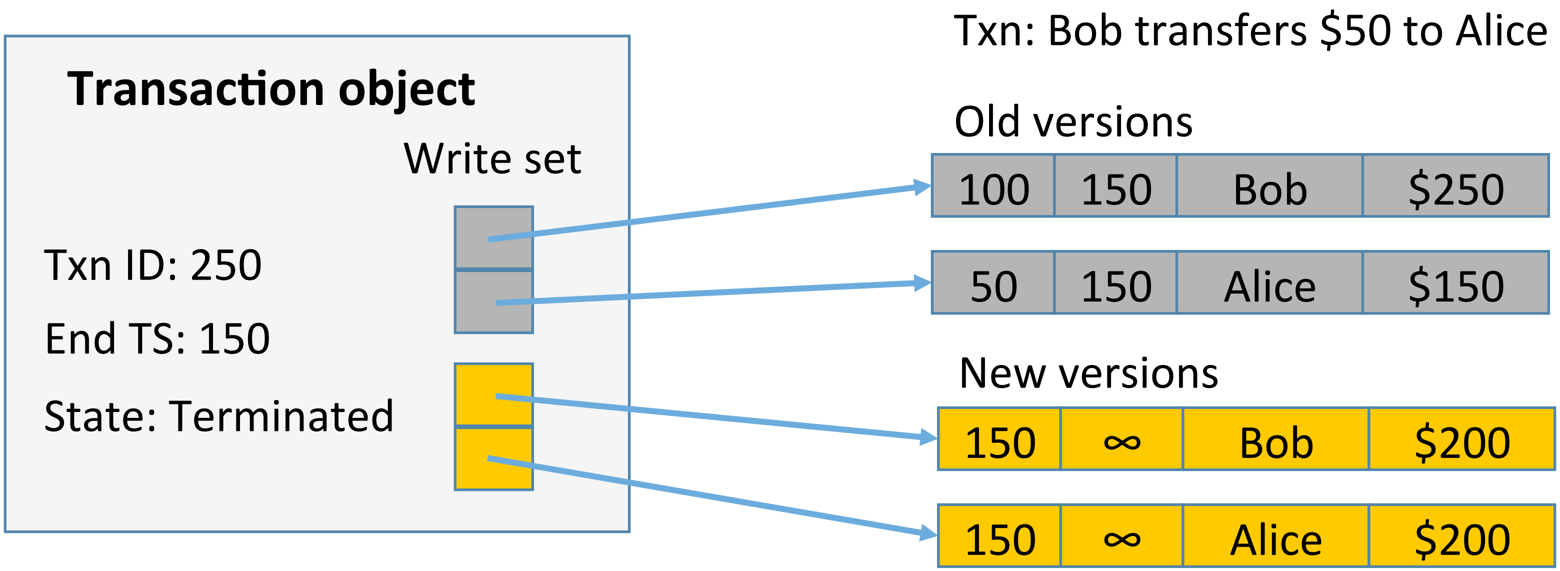


Query and transaction interop

- Regular SQL queries can access Hekaton tables like any other table
  - Slower than through a compiled stored procedure
- A query can mix Hekaton tables and regular SQL tables
- A transaction can update both SQL and Hekaton tables

- Crucial feature for customer acceptance
  - Greatly simplifies application migration
  - Feature completeness – any query against Hekaton tables
  - Ad-hoc queries against Hekaton tables
  - Queries and transactions across SQL and Hekaton tables

When can old versions be discarded?



- Can discard the old versions as soon as the read time of the oldest active transaction is over 150
- Old versions easily found – use pointers in write set
- Two steps: unhook version from all indexes, release record slot

Hekaton garbage collection

- **Non-blocking** – runs concurrently with regular processing
- **Cooperative** – worker threads remove old versions when encountered
- **Incremental** – small batches, can be interrupted at any time
- **Parallel** -- multiple threads can run GC concurrently
- **Self-throttling** – done by regular worker threads in small batches

- Overhead depends on read/write ratio
  - Measured 15% overhead on a very write-heavy workload
  - Typically much less

Durability and availability

- **Logging** changes before transaction commit
  - All new versions, keys of old versions in a single IO
  - Aborted transactions write nothing to the log
- **Checkpoint** - maintained by rolling log forward
  - Organized for fast, parallel recovery
  - Require only sequential IO
- **Recovery** – rebuild in-memory database from checkpoint and log
  - Scan checkpoint files (in parallel), insert records, and update indexes
  - Apply tail of the log
- **High availability (HA)** – based on replicas and automatic failover
  - Integrated with AlwaysOn (SQL Server's HA solution)
  - Up to 8 synch and asynch replicas
  - Can be used for read-only queries

CPU efficiency for lookups

Transaction size in #lookups	CPU cycles (in millions)		Speedup
	Interpreted	Compiled	
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

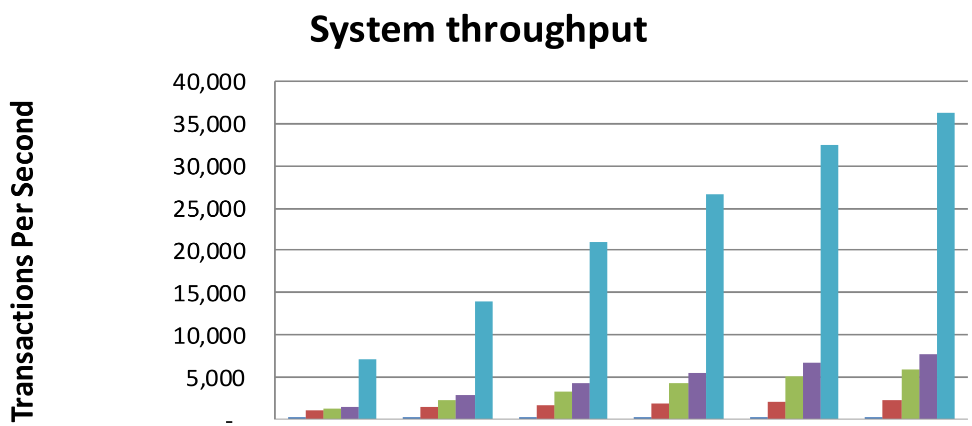
- Random lookups in a table with 10M rows
- All data in memory
- Intel Xeon W3520 2.67 GHz
- Performance: 2.7M lookups/sec/core

CPU efficiency for updates

Transaction size in #updates	CPU cycles (in millions)		Speedup
	Interpreted	Compiled	
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

- Random updates, 10M rows, one index, snapshot isolation
- Log writes disabled (disk became bottleneck)
- Intel Xeon W3520 2.67 GHz
- Performance: 1.9M updates/sec/core

Throughput under high contention



	2	4	6	8	10	12
Number of cores	2	4	6	8	10	12
SQL with contention	984	1,363	1,645	1,876	2,118	2,312
SQL without contention	1,153	2,157	3,161	4,211	5,093	5,834
Interop	1,518	2,936	4,273	5,459	6,701	7,709
Native	7,078	13,892	20,919	26,721	32,507	36,375

Throughput improvements

- Converting table but using interop
  - 3.3X higher throughput
- Converting table and stored proc
  - 15.7X higher throughput

- Workload: read/insert into a table with a unique index
- Insert txn (50%): append a batch of 100 rows
- Read txn (50%): read last inserted batch of rows

Initial customer experiences

- **Bwin** – large online betting company
- Application: session state
  - Read and updated for every web interaction
- Current max throughput: 15,000 requests/sec
- Throughput with Hekaton: 250,000 requests/sec

- **EdgeNet** – provides up-to-date inventory status information
- Application: rapid ingestion of inventory data from retailers
- Current max ingestion rate: 7,450 rows/sec
- Hekaton ingestion rate: 126,665 rows/sec
- Allows them to move to continuous, online ingestion from once-a-day batch ingestion

- **SBI Liquidity Market** – foreign exchange broker
- Application: online calculation of currency prices from aggregate trading data
- Current throughput: 2,812 TPS with 4 sec latency
- Hekaton throughput: 5,313 TPS with <1 sec latency

Status

- Hekaton will ship in SQL Server 2014
- SQL Server 2014 to be released early in 2014
- Second public beta (CTP2) available now

Thank you for your attention.

Questions?