Main-Memory Databases
Motivation

Hardware trends

- Huge main memory capacity with complex access characteristics (Caches, NUMA)
- Many-core CPUs
- SIMD support in CPUs
- New CPU features (HTM)
- Also: Graphic cards, FPGAs, low latency networking, . . .

Database system trends

- Entire database fits into main memory
- New types of database systems
- New algorithms, new data structures

“The End of an Architectural Era. (It’s Time for a Complete Rewrite).”
Recap: Database Workloads

Analytics

- Long-running
- Access large parts of the database
- Often use scans
- Read-only
- Example: “Average order value per year and product group?”

Transaction processing

- Short running
- (Multiple) point queries + simple control flow
- Insert/Update/Delete/Read data
- Example: “Increment account x by 10, decrement account y by 10”

Universal DBMS used for both (but not concurrently).
OLTP

Universal DBMS were optimized for 1970’s hardware
- Small fraction of DB in memory buffer
- Hide and avoid disk access at any cost

Today
- Even enterprises can store entire DB in memory
- Transaction are often “one-shot”
- Transactions execute in a few ms or even µs
OLTP (2)

Main sources of overhead

- ARIES-style logging
- Locking (2PL)
- Latching
- Buffer Management

Useful work can be as low as $\frac{1}{60}$th of instructions\(^1\). Modern systems avoid this overhead (see slide 9).

\(^1\)Harizopoulos et al. – *OLTP Through the Looking Glass, and What We Found There*
Physical Data Layout in Main Memory

Lightweight:
- Buffer Manager removed
- No need for segments
- No need for slotted pages

Store data in simple arrays. But: Row-wise or column-wise?
Physical Data Layout in Main Memory (2)

Row Store:
- Beneficial when accessing many attributes
- For OLTP

Column Store:
- Excellent cache utilization
- Sometimes individually sorted
- Compression potential
- Vectorized processing
- For OLAP

Hybrid Row/Column Stores possible
New Systems (Examples)

OLTP-only:
- VoltDB/H-Store
- Microsoft Hekaton

OLAP-only:
- Vectorwise
- MonetDB
- DB2 BLU

Hybrid OLTP and OLAP:
- SAP HANA
- HyPer
New Systems: OLTP (Examples)

Challenge:

- Avoid overhead
- Guarantee ACID

Approaches:

- Buffer Management: Removed
- Logging
  - H-Store/VoltDB: Log shipping to other nodes
  - Hekaton: Lightweight logging (no index structures)
- Locking:
  - H-Store/VoltDB: Serial execution (on private partitions)
  - Hekaton: Optimistic MVCC
- Latching
  - H-Store/VoltDB: Not necessary
  - Hekaton: Latch-free data structures
New Systems: Hekaton

- Integrated in SQL Server
- Code Generation
- Only access path: Index (Hash or B(w)-Tree)
- Latch-Free Indexes
- MVCC
New Systems: OLAP

- Vectorwise: Vectorized Processing
- HyPer: Query Compilation (cf. Chapter *Code Generation*)
Modern Hardware

Main-Memory Databases

New Systems: Hybrid OLTP and OLAP

Traditionally:

- Mixing OLTP and OLAP leads to performance decline
- ETL architecture
- 2 systems, stale data

New Systems

- SAP HANA
  - Split DB into read-optimized *main* and update-friendly *delta*
  - OLAP queries read main, OLTP transactions read delta *and* main
  - Periodically merge main and delta

- HyPer: Virtual memory snapshots
HyPer: Virtual Memory Snapshots

OLTP Data

OLTP Tx

A   C
B   D
E   G
F   H
HyPer: Virtual Memory Snapshots
HyPer: Virtual Memory Snapshots

forked OLAP-Snapshot

OLTP Data

OLTP Tx

OLAP Queries

read C
HyPer: Virtual Memory Snapshots
HyPer: Virtual Memory Snapshots
In-Memory Index Structures

- In-memory hash indexes
  - Simple and fast
  - Growing is very expensive
  - Do not support range queries

- Search Trees
  - BSTs are cache unfriendly
  - B-Trees better (even though designed for disk)

- Radix-Trees ("Tries")
  - Support range queries
  - Height is independent from number of entries
Radix Trees

Properties:
- Height depends on key length, not number of entries
- No rebalancing
- All insertion orders yield same tree
- Keys are stored in the tree implicitly

Search:
- Node is array of size $2^s$
- $s$ bits (often 8) are used as an index into the array
- $s$ is a trade-off between lookup-performance and memory consumption
Adaptive Radix Trees

Four node types:

- **Node4**: 4 keys and 4 pointers at corresponding positions:

  ![Node4 Diagram]

  - Key positions: 0, 1, 2, 3
  - Pointer positions: 0, 1, 2, 3

  - Keys: 0, 2, 3, 255
  - Pointers: a, b, c, d

- **Node16**: Like Node4, but with 16 keys. SIMD searchable.

- **Node48**: Full 256 keys (index offset), point to up to 48 values:

  ![Node48 Diagram]

  - Child index positions: 0, 1, 2, 3
  - Child pointer positions: 0, 1, 2

  - Keys: 0, 2, 3, 255
  - Pointers: b, a, c, d

- **Node256**: Regular trie node, i.e. array of size 256

  Additionally: Header with node type, number of entries.
Exploiting HTM for OLTP

- Intel’s Haswell introduced HTM (via cache coherency protocol)
- Allows to group instructions to transactions
- Can help to implement DB transactions, but
  - Do not guarantee ACID by themselves
  - Limited in size/time

⇒ Use HTM transactions as building blocks for DB transactions
Exploiting HTM for OLTP (2)

Goals:
- As fine-grained as 2PL, but faster
- As fast as serial execution, but more flexible

atomic-elide-lock (lock) {
    account[from] -= amount;
    account[to] += amount;
}
Implementing DB transactions with HTM

Use TSO + HTM for latching:

- **Database transaction**
  - conflict detection: read/write sets via timestamps
  - elided lock: serial execution
  - request a new timestamp, record safe timestamp

- **HTM transaction**
  - conflict detection: read/write sets in hardware
  - elided lock: latch
  - single tuple access
  - verify/update tuple timestamps

- **HTM transaction**
  - conflict detection: read/write sets in hardware
  - elided lock: latch
  - single tuple access
  - verify/update tuple timestamps

- **HTM transaction**
  - conflict detection: read/write sets in hardware
  - elided lock: latch
  - single tuple access
  - verify/update tuple timestamps

- release timestamp, update safe timestamp

- Relation and index structure layout must avoid conflicts
NUMA-Aware Data Processing

NUMA architectures:

- Local access cheap
- Remote access expensive
NUMA-Aware Data Processing: Hash Join

Phase 1: process T morsel-wise and store NUMA-local storage area.

Phase 2: scan NUMA-local storage area and insert pointers into HT.

Insert the pointer into HT.

...((T))...

σ(T) σ(T) σ(T)

Storage area of red core

Storage area of green core

Storage area of blue core

HT(T)

HT(S)

σ(R) σ(R) σ(R)

Next morsel
Compaction

- OLTP & OLAP share the same physical data model
  - Fast modifications vs scan performance
  - Row store vs column store
- Modifications require snapshot maintenance
  - Use more memory
  - Congest memory bus
  - Stall transactions
Compaction: Hot/Cold Clustering

- Compression is applied asynchronously to cold part:
  - Dictionary encoding
  - Run-length encoding
  - Other schemes possible

- Compact snapshots through a mix of regular and huge pages
  - Keeps page table small
  - Clustered updates
  - No huge pages need to be replicated
Compaction: Hot/Cold Clustering

**Cooling**
- Hot & cold items mixed
- Uncompressed
- Small memory pages

**Hot**
- Working Set (hot data)
- Uncompressed
- Small memory pages

**Cold**
- Cold data items only
- Not compressed yet

**Frozen**
- Cold & compressed data
- Huge memory pages
- Rarely accessed by OLTP
- Immutable: Deleted and updated items are marked "invalid" and copied to Hot

"Invalid frozen items" data structure
Compaction: Hot/Cold Clustering

How to detect temperature without causing overhead?

1. Software: LRU lists, counters
2. Hardware: mprotect
3. Hardware: dirty and young flags