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?

Logical Table

<table>
<thead>
<tr>
<th>EmpId</th>
<th>First</th>
<th>Last</th>
<th>Salery</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Joe</td>
<td>Doe</td>
<td>40000</td>
</tr>
<tr>
<td>20</td>
<td>Mary</td>
<td>Jones</td>
<td>32000</td>
</tr>
<tr>
<td>32</td>
<td>Bob</td>
<td>Black</td>
<td>60000</td>
</tr>
<tr>
<td>25</td>
<td>Jane</td>
<td>Jones</td>
<td>85000</td>
</tr>
</tbody>
</table>

Column Store

Row Store
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*)
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
HyPer: Virtual Memory Snapshots

forked OLAP-Snapshot

OLTP Data

OLTP Tx

A C
B D
E G
F H

OLAP Queries
HyPer: Virtual Memory Snapshots

[Diagram showing OLTP Data, forked OLAP-Snapshot, OLAP Queries, OLTP Tx, reading C]
HyPer: Virtual Memory Snapshots

forked OLAP-Snapshot

OLTP Data

update C to C*

OLAP Queries

C

D

A
B
C*

E
F
G
H

copy-on-write

OLTP Tx
HyPer: Virtual Memory Snapshots

forked OLAP-Snapshot

OLTP Data

OLTP Tx

OLAP Queries

read C

read H

C* D

A B C* D

E F G H
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]

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

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

  ![Node48 Diagram]

  - 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:

- 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 of red core

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

- Storage area of red core
- Storage area of green core
- Storage area of blue core

**HT(T):**
- Storage area of blue core
- Storage area of green core
- Storage area of red core

**HT(S):**
- Storage area of blue core
- Storage area of green core
- Storage area of red core

**HT(I):**
- Global Hash Table

Insert the pointer into HT

... (T) ... (T) ... (T)
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

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

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

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

"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
Data Blocks

- most data is cold and rarely / never changes
- it is attractive to compress these aggressively
- and pre-compute SMAs
- helps with skipping data
- fits well with a cloud storage setup
Data Blocks - Scan Types

- **uncompressed chunk**: Vectors of e.g., 8192 records
- **compressed Data Block**: Vectorized evaluation of SARGable predicates on compressed data and unpacking of matches
- **interpreted vectorized scan on Data Block**: Vectorized evaluation of SARGable predicates and copying of matches
- **JIT-compiled tuple-at-a-time query pipeline**: Tuple-at-a-time evaluation of scan predicates
- **uncompressed chunk**: JIT-compiled tuple-at-a-time scan on uncompressed chunk

Index

PSMAs

A B C D

vector

A B D

tuple

push matches tuple-at-a-time

single interface

hot scan

cold scan
## Data Blocks - Layout

<table>
<thead>
<tr>
<th>tuple count</th>
<th>sma offset₀</th>
<th>dict offset₀</th>
<th>data offset₀</th>
</tr>
</thead>
<tbody>
<tr>
<td>compression₀</td>
<td>string offset₀</td>
<td>sma offset₁</td>
<td>dict offset₁</td>
</tr>
<tr>
<td>data offset₁</td>
<td>compression₁</td>
<td>string offset₁</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>sma offsetₙ</td>
<td>dict offsetₙ</td>
<td>data offsetₙ</td>
</tr>
<tr>
<td>compressionₙ</td>
<td>string offsetₙ</td>
<td>min₀</td>
<td>max₀</td>
</tr>
</tbody>
</table>

**lookup table₀**

**Positional SMA index for attribute 0**

<table>
<thead>
<tr>
<th>domain size₀</th>
<th>dictionary₀</th>
</tr>
</thead>
</table>

**compressed data₀**

**string data₀**

| min₁ | max₁ | ... |
Data Blocks - Vectorized Evaluation

- aligned data
- unaligned data
- read offset
- predicate evaluation
- movemask
  \[=154_{10}\]
- remaining data
- precomputed positions table
- data
- lookup
  \[0, 3, 4, 6\]
- add global scan position and update match vector
- write offset
  \[0, 1, 2, 3, 4, 5, 6, 7\]
- match positions
  \[1, 3, 5, 7, 9, 11, 14, 15, 17\]