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)
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

<table>
<thead>
<tr>
<th>OLTP Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
</tr>
<tr>
<td>B</td>
</tr>
<tr>
<td>C</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OLTP Tx</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
</tr>
<tr>
<td>F</td>
</tr>
<tr>
<td>G</td>
</tr>
<tr>
<td>H</td>
</tr>
</tbody>
</table>
HyPer: Virtual Memory Snapshots

OLTP Data

forked OLAP-Snapshot

OLAP Queries

OLTP Tx
HyPer: Virtual Memory Snapshots
HyPer: Virtual Memory Snapshots
HyPer: Virtual Memory Snapshots

forked OLAP-Snapshot

OLTP Data

OLTP Tx

OLAP Queries

read C

read H

C*

D

A

B

G

H

E

F

C

D

OLTP Data

OLAP Queries

forked OLAP-Snapshot

read C

read 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]

  ```plaintext
  key | child pointer
  0   | 2   | 3   | 255
  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]

  ```plaintext
  child index | child pointer
  0  1  2  3 | 255 0  1  2  47
  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

```c
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 conflict

- 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

Global Hash Table

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)}

Storage area of red core

Storage area of green core

Storage area of blue core

Scan into HT

Insert the pointers into HT

σ(HT(T)) σ(HT(S))
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

Huge memory page
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

- **Interpreted Vectorized Scan on Data Block**
  - Vectorized evaluation of SARGable predicates on compressed data and unpacking of matches.
  - Vectors of e.g., 8192 records.

- **Interpreted Vectorized Scan on Uncompressed Chunk**
  - Vectorized evaluation of SARGable predicates and copying of matches.

- **JIT-Compiled Tuple-at-a-Time Query Pipeline**
  - Push matches tuple-at-a-time.

- **Cold Scan**
  - Tuple-at-a-time evaluation of scan predicates.

- **Hot Scan**
  - Tuple-at-a-time single interface.
# Data Blocks - Layout

<table>
<thead>
<tr>
<th>tuple count</th>
<th>sma offset_0</th>
<th>dict offset_0</th>
<th>data offset_0</th>
</tr>
</thead>
<tbody>
<tr>
<td>compression_0</td>
<td>string offset_0</td>
<td>sma offset_1</td>
<td>dict offset_1</td>
</tr>
<tr>
<td>data offset_1</td>
<td>compression_1</td>
<td>string offset_1</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>sma offset_n</td>
<td>dict offset_n</td>
<td>data offset_n</td>
</tr>
</tbody>
</table>

**Positional SMA index for attribute 0**

<table>
<thead>
<tr>
<th>domain size_0</th>
<th>dictionary_0</th>
</tr>
</thead>
<tbody>
<tr>
<td>compressed data_0</td>
<td></td>
</tr>
<tr>
<td>string data_0</td>
<td></td>
</tr>
<tr>
<td>min_1</td>
<td>max_1</td>
</tr>
</tbody>
</table>
Data Blocks - Vectorized Evaluation

- aligned data
- unaligned data
- read offset
- remaining data
- predicate evaluation
- movemask
- precomputed positions table
- add global scan position and update match vector
- write offset
- match positions

precomputed positions table:
- 0, 1, 2, 3, 4, 5, 6, 7

lookup:
- 0, 3, 4, 6

movemask = 154_{10}

add global scan position and update match vector

write offset

match positions:
- 1, 3, 5, 7, 9, 11, 14, 15, 17

lookup:
- 0, 1, 2, 3, 4, 5, 6, 7

precomputed positions table:
- 0, 3, 4, 6

movemask = 154_{10}

add global scan position and update match vector

write offset

match positions:
- 1, 3, 5, 7, 9, 11, 14, 15, 17