New/Main/in-memory Database Systems
Proloferation of NEW DB(-like) Implementations

Quick sample:

... and dozens of others

In contrast to...

... before Y~2000 it was pretty much divided between the big four (ORACLE, IBM/DB2, Sybase, and MS Server) and (later, with the advent of the WEB) Postgress, MySQL, etc.
Prolofection of **NEW DB(-like) Implementations**

Quick sample:

... and dozens of others

In contrast to...

... before \( Y \approx 2000 \) it was pretty much divided between the *big four* (ORALE, IBM/DB2, Sybase, and MS Server) and (later, with the advent of the WEB) Postgress, MySQL, etc.
New Circumstances

1. cheap and abundant hardware (incl. Main Memory)
2. changes in applications/workloads
3. cost (we won’t focus on this though)
Why so many? And why Main/In-Memory? (M/IMDB)

New Circumstances

1. cheap and abundant hardware (incl. Main Memory)
2. changes in applications/workloads
3. cost (we won’t focus on this though)
Requirements (user point of view)

Goal of a DBMS
Execute user queries/Updates (as fast as possible)

(typical) Requirements:

1. Stores all of your Data (scalability)
2. Physical Data Independence (SQL vs. B-trees et al.)
3. Durability (the idea of a transaction)
4. Isolation (sharing/concurrency)

⇒ do we need all of the above all the time?
Requirements (user point of view)

Goal of a DBMS
Execute user queries/Updates (as fast as possible)

(typical) Requirements:

1. Stores all of your Data (scalability)
2. Physical Data Independence (SQL vs. B-trees et al.)
3. Durability (the idea of a transaction)
4. Isolation (sharing/concurrency)

⇒ do we need all of the above all the time?
Requirements (user point of view)

Goal of a DBMS

Execute user queries/Updates (as fast as possible)

(typical) Requirements:

1. Stores all of your Data (scalability)
2. Physical Data Independence (SQL vs. B-trees et al.)
3. Durability (the idea of a transaction)
4. Isolation (sharing/concurrency)

⇒ do we need all of the above all the time?
Standard Architecture: Client-Server “System”

Query/Update Compiler
⇒ compiles a *logical expression* to a *plan*

Query/Update Execution Engine:
⇒ executes a *prepared plan*

1. Query processor (access paths)
2. Transaction Manager
3. Recovery Manager
4. Buffer Pool
Where does the Time go? (a case study)

- SHORE (Scalable Heterogeneous Object Repository, Wisconsin ’90s)
  ⇒ the whole database is preloaded in main memory
- TPC-C (OLTP) benchmark: “new order” and “payment” transactions
  ⇒ 50/50 mix of the transactions in experiments

- Experiments show performance gain by removing/simplifying:
  1. B-Tree keys (no prefix compression)
  2. no logging (no durability)
  3. no locks (no concurrency)
  4. no latches (no transactions: begin/commit/…)
  5. no buffer manager (remember DB preloaded!)
Where does the Time go? (a case study)

- SHORE (Scalable Heterogeneous Object Repository, Wisconsin ’90s) ⇒ the whole database is preloaded in main memory
- TPC-C (OLTP) benchmark: “new order” and “payment” transactions

![Figure 3. TPC-C Schema.](image-url)
Where does the Time go? (a case study)

- SHORE (Scalable Heterogeneous Object Repository, Wisconsin ’90s) ⇒ the whole database is preloaded in main memory
- TPC-C (OLTP) benchmark: “new order” and “payment” transactions ⇒ 50/50 mix of the transactions in experiments
- Experiments show performance gain by removing/simplifying:
  1. B-Tree keys (no prefix compression)
  2. no logging (no durability)
  3. no locks (no concurrency)
  4. no latches (no transactions: begin/commit/...)
  5. no buffer manager (remember DB preloaded!)

(University of Waterloo)
Where does the Time go? (setup)

Assumptions:

1. all data preloaded into main memory
2. transactions compiled and linked against SHORE
3. 50-50 mix
4. 40k transaction runs

Figure 4. Calls to Shore’s methods for New Order and Payment transactions.
Where does the Time go?

Given these basic throughput measurements, we now give detailed instruction breakdowns for the two transactions of our benchmark. Recall that the instruction and cycle breakdowns in the following sections do not include any impact of disk operations, whereas the throughput numbers for baseline Shore do include some log write operations.

4.3.2 Payment

Figure 5 (left side) shows the reductions in the instruction count of the Payment transaction as we optimized B-tree key evaluations and removed logging, locking, latching, and buffer manager functionality. The right part of the figure shows, for each feature removal we perform, its effect on the number of instructions spent in various portions of the transaction's execution. For the Payment transaction, these portions include a begin call, three B-tree lookups followed by three pin/unpin operations, followed by three updates (through the B-tree), one record creation and a commit call. The height of each bar is always the total number of instructions executed. The right-most bar is the performance of our minimal-overhead kernel.

Our B-tree key evaluation optimizations are reportedly standard practice in high-performance DBMS architectures, so we perform them first because any system should be able to do this.

Removing logging affects mainly commits and updates, as those are the portions of the code that write log records, and to a lesser degree B-tree and directory lookups. These modifications remove about 18% of the total instruction count.

Locking takes the second most instructions, accounting for about 25% of the total count. Removing it affects all of the code, but is especially important in the pin/unpin operations, the lookups, and commits, which was expected as these are the operations that must acquire or release locks (the transaction already has locks on the updated records when the updates are performed).

Figure 5. Detailed instruction count breakdown for Payment transaction.
Given these basic throughput measurements, we now give detailed instruction breakdowns for the two transactions of our benchmark. Recall that the instruction and cycle breakdowns in the following sections do not include any impact of disk operations, whereas the throughput numbers for baseline Shore do include some log write operations.

4.3.2 Payment

Figure 5 (left side) shows the reductions in the instruction count of the Payment transaction as we optimized B-tree key evaluations and removed logging, latching, and buffer manager functionality. The right part of the figure shows, for each feature removal we perform, its effect on the number of instructions spent in various portions of the transaction's execution. For the Payment transaction, these portions include a begin call, three B-tree lookups followed by three pin/unpin operations, followed by three updates (through the B-tree), one record creation and a commit call. The height of each bar is always the total number of instructions executed. The right-most bar is the performance of our minimal-overhead kernel.

Our B-tree key evaluation optimizations are reportedly standard practice in high-performance DBMS architectures, so we perform them first because any system should be able to do this. Removing logging affects mainly commits and updates, as those are the portions of the code that write log records, and to a lesser degree B-tree and directory lookups. These modifications remove about 18% of the total instruction count. Locking takes the second most instructions, accounting for about 25% of the total count. Removing it affects all of the code, but is especially important in the pin/unpin operations, the lookups, and commits, which was expected as these are the operations that must acquire or release locks (the transaction already has locks on the updated records when the updates are performed).

Figure 6. Detailed instruction count breakdown for New Order transaction.
Where does the Time go?

Figure 8. Instructions (left) vs. Cycles (right) for New Order.
Where does the Time go? (conclusions)

Having a giant buffer cache to fit the whole dataset doesn’t seem to solve all problems (90+% OVERHEAD!)

However...

...the savings in experiments at cost of functionality

⇒ can MMDBs be engineered to mitigate the overhead without sacrificing functionality?

- Single threading vs. multicore
- Availability (replication) vs. logging
- Variations on isolation
- Cache-conscious data structures

(University of Waterloo)
Where does the Time go? (conclusions)

Having a giant buffer cache to fit the whole dataset doesn’t seem to solve all problems (90+% OVERHEAD!)

However...

...the savings in experiments at cost of functionality

⇒ can MMDBs be engineered to mitigate the overhead without sacrificing functionality?

- Single threading vs. multicore
- Availability (replication) vs. logging
- Variations on isolation
- Cache-conscious data structures
Where does the Time go? (conclusions)

Having a giant buffer cache to fit the whole dataset doesn’t seem to solve all problems (90+% OVERHEAD!)

However...

...the savings in experiments at cost of functionality

⇒ can MMDBs be engineered to mitigate the overhead without sacrificing functionality?

- Single threading vs. multicore
- Availability (replication) vs. logging
- Variations on isolation
- Cache-conscious data structures
Take Home

Lots of open issues:

1. DB engine vs. Compilation approaches
2. Main memory data organization (multilevel memory)
3. Taking advantage of parallelism (many levels)
Take Home

Lots of open issues:

1. DB engine vs. Compilation approaches
2. Main memory data organization (multilevel memory)
3. Taking advantage of parallelism (many levels)
4. ...