Topics in Database Systems:
Main/in-memory and Embedded DBMS
CS848 Spring 2018

David Toman

DATABASE IMPLEMENTATION
(OVERVIEW OF STANDARD TECHNIQUES)
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:

- B-Tree keys (no prefix compression)
- no logging (no durability)
- no locks (no concurrency)
- no latches (no transactions: begin/commit/…)
- 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

![TPC-C Schema](image)

**Figure 3. TPC-C Schema.**
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? (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.
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).
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, 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.

---

Figure 6. Detailed instruction count breakdown for New Order transaction.

---

### 4.7 New Order

Figure 6 (left side) shows the reductions in the instruction count of the New Order 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 New Order transaction, these portions include a begin call, 13 x insert index followed by 12 x create record, 23 x pin/unpin, 23 x B-tree lookup, 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
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
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 (ORALE, IBM/DB2, Sybase, and MS Server)

and (later, with the advent of the WEB) Postgress, MySQL, etc.
Proloferation of **NEW DB(-like) Implementations**

Quick sample:

... and dozens of others

In contrast to...

... before $Y \sim 2000$ 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.
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)
Compilation-based Approaches

IDEA:

can we use a

- high level system *description*
- a compiler

to generate *tailored* code for our application?
Compilation-based Approaches

Definability and Rewriting

<table>
<thead>
<tr>
<th>Queries</th>
<th>range-restricted FOL (a.k.a. SQL)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ontology/Schema</td>
<td>range-restricted FOL $\Sigma := \Sigma^L \cup \Sigma^{LP} \cup \Sigma^P$</td>
</tr>
<tr>
<td>Data</td>
<td>CWA (complete information)</td>
</tr>
</tbody>
</table>

$$\Sigma_L \xrightarrow{\phi} S_L \xrightarrow{\psi} Compiler \psi(Relational Algebra over SA) \xrightarrow{\Sigma} Evaluator \rightarrow Answers$$

Data CWA (complete information)
Compilation-based Approaches

Definability and Rewriting

<table>
<thead>
<tr>
<th>Queries</th>
<th>range-restricted FOL over $S_L$ definable w.r.t. $\Sigma$ and $S_A$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ontology/Schema</td>
<td>range-restricted FOL $\Sigma := \Sigma^L \cup \Sigma^{LP} \cup \Sigma^P$</td>
</tr>
<tr>
<td>Data</td>
<td>CWA (complete information for $S_A$ symbols)</td>
</tr>
</tbody>
</table>

Compilation-based Approaches

Definability and Rewriting

<table>
<thead>
<tr>
<th>Queries</th>
<th>range-restricted FOL over $S_L$ definable w.r.t. $\Sigma$ and $S_A$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ontology/Schema</td>
<td>range-restricted FOL $\Sigma := \Sigma^L \cup \Sigma^{LP} \cup \Sigma^P$</td>
</tr>
<tr>
<td>Data</td>
<td>CWA (complete information for $S_A$ symbols)</td>
</tr>
</tbody>
</table>

- to users it looks like a *single model* (of the logical schema)
- implementation can pick from many models
  but *definable* queries answer the same in each of them

![Diagram]

- **Query** $(S_L)$
- **Schema** $(S_L \cup S_P)$
- **Data** $(S_A \subseteq S_P)$
- **Compiler**
  - $\psi$ (Relational Algebra over $S_A$)
- **Evaluator**
  - $\Sigma$ (instance of) $S_A$
- **Answers**
Compilation-based Approaches

Definability and Rewriting

Queries: range-restricted FOL over $S_L$
Ontology/Schema: range-restricted FOL $\Sigma := \Sigma^L$
Data: CWA (complete information for $S_A$)

- to users it looks like a single model (of the logical schema)
- implementation can pick from many models but definable queries answer

$$\psi$$

Query ($S_L$)

Compiler

$\psi$ (Relational Algebra over $S_A$)

Evaluator

$\Sigma$

Data ($S_A \subseteq S_P$)

Answers

(instance of) $S_A$
The LINUX-INFO System: A Case Study

GOAL:

to develop the LINUX-INFO system to monitor the operating systems deployed in their organization.

CONSEQUENCES:

code (parts) of Linux KERNEL (e.g., entries in the /proc filesystem)
The LINUX-INFO System: A Case Study

GOAL:
to develop the LINUX-INFO system to monitor the operating systems deployed in their organization.

CONSEQUENCES:
code (parts) of Linux KERNEL (e.g., entries in the /proc filesystem)
Example of LINUX-INFO data important to APS.

1. process `gcc` is running
2. `gcc`’s process number is 1234.
3. the user running `gcc` is 145.
4. `gcc` uses file “foo.c”

Example of LINUX-INFO metadata specified by APS.

1. There entities called process and file.
2. There are attributes called pno, pname, uname, and fname.
3. Each process entity has attributes pno, pname and uname.
4. Each file entity has attribute fname.
5. Processes are identified by their pno.
6. Files are identified by their fname.
7. There is a relationship uses between processes and files.
**LINUX-INFO System: Data and Metadata**

Example of LINUX-INFO data important to APS.

1. process `gcc` is running
2. `gcc`’s process number is 1234.
3. the user running `gcc` is 145.
4. `gcc` uses file “foo.c”

Example of LINUX-INFO metadata specified by APS.

4. There entities called `process` and `file`.
5. There are attributes called `pno`, `pname`, `uname`, and `fname`.
6. Each process entity has attributes `pno`, `pname` and `uname`.
7. Each file entity has attribute `fname`.
8. Processes are identified by their `pno`.
9. Files are identified by their `fname`.
10. There is a relationship `uses` between processes and files.
A physical design for LINUX (selected by Linus Torvalds).

8 There are process records called task-struct.

9 Each task-struct record has record fields pid, uid, comm, and file-struct.

10 All task-structs is organized as a tree data structure.

11 The task-struct records correspond one-to-one to process entities.

12 Record fields in task-struct encode the corresponding attribute values for process entities, for example, pid encodes an pno, etc.

13 Similarly, fs's correspond appropriately to (open) file entities.

14 file-struct field of task-struct is an array of fds; an entry in this array indicates that the process corresponding to this task-struct is using the file represented by the fd record in the array.
A LINUX-INFO user query specified by APS.

14 Find the files used by process invoked by user 145.

A query plan selected by a query compiler.

15 Scan tree of task-structs, for each check if its uid attribute is 145 and, if so scan the file-struct array in the task-struct and print out the names of files described by non-NULL file descriptors (fd).

Question:

Does the physical design allow APS to list all files known to the Linux system?
A **LINUX-INFO user query** specified by APS.

14 Find the **files used by process** invoked by user 145.

A **query plan** selected by a **query compiler**.

15 Scan tree of **task-structs**, for each check if its **uid** attribute is 145 and, if so scan the **file-struct array** in the **task-struct** and print out the names of files described by non-NULL file descriptors (**fd**).

**Question:**

Does the **physical design** allow APS to list all files known to the Linux system?
A **Linux-Info** *user query* specified by APS.

14 Find the *files used by process invoked by user 145*.

A *query plan* selected by a *query compiler*.

15 Scan tree of *task-structs*, for each check if its *uid* attribute is 145 and, if so scan the *file-struct array in the task-struct* and print out the names of files described by non-NULL file descriptors (*fd*).

**Question:**

Does the *physical design* allow APS to list all files known to the Linux system?
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)
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. ...