Topics in Database Systems: Modern Database Systems
CS848 Spring 2022

David Toman

Wednesday 10:30-1:00 (DC 2568)
cs.uwaterloo.ca/~david/cs848/
Proliferation of NEW DB(-like) Implementations

Quick sample:

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.

... and dozens of others
Proliferation 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.
Why so many? And why Main-Memory?

New Circumstances

1. cheap and abundant hardware (Extra CPUs and Main Memory)
2. changes in applications/workloads (often fit in main memory!)
3. cost (we won’t focus on this though)
Why so many? And why Main-Memory?

New Circumstances

1. cheap and abundant hardware (Extra CPUs and Main Memory)
2. changes in applications/workloads (often fit in main memory!)
3. cost (we won’t focus on this though)
Topics of Interest

1. What are the main differences between managing memory resident data v.s. data in external storage?
   - impact on query/update processing
     how many *instructions* does it take to answer simple queries?
   - what happens to ACID (and can we afford durability at all)?

2. What is the impact on *programming interface* to MMDBs?
   - declarative (SQL-like) vs. procedural (C++-like)
   - query optimization?

3. What is the impact of multi-core/CPU hardware
   - data partitioning and query compilation/allocation
   - communication/synchronization between parallel operations
     dependency on architecture (Multicore, NUMA, Shared-nothing)?

4. UDTs (user-defined topics)
Topics of Interest

1. What are the main differences between managing memory resident data v.s. data in external storage?
   - impact on query/update processing
   - how many instructions does it take to answer simple queries?
   - what happens to ACID (and can we afford durability at all)?

2. What is the impact on programming interface to MMDBs?
   - declarative (SQL-like) vs. procedural (C++-like)
   - query optimization?

3. What is the impact of multi-core/CPU hardware
   - data partitioning and query compilation/allocation
   - communication/synchronization between parallel operations
   - dependency on architecture (Multicore, NUMA, Shared-nothing)?

4. UDTs (user-defined topics)
Outline & Organization

• Organization:
  ⇒ Lectures (4-5),
  ⇒ Presentations of papers (reading list), and
  ⇒ Projects

• First meeting: Wed May 4, 2022 at 10:30 in DC 2568

• Prerequisites:
  ⇒ *Intro to Databases* (CS348-like), and
  ⇒ standard programming skills
    (although this is not necessarily an implementation class)

• Class web site: [cs.uwaterloo.ca/~david/cs848/](http://cs.uwaterloo.ca/~david/cs848/)
  reading list, schedule of classes/presentations, policies, etc.
Organization (ii)

Week 1: Organization,
Issues in *classical* DB implementations, and
What can be done about it?

Week 2: Introduction to DB implementation,
Classical Approaches vs. Query compilation (examples);
Discussion/assignment of presentations/projects.

Weeks 3-5: More on Query Compilation:
Multi-level Store (a.k.a. Disks),
Sorted Data and better algorithms,
How does this really work?
What to do about Updates? (and perhaps more)

Weeks 7-12: In-class Paper/Project Discussion&Consultation

Week 13: Summary and Wrap-up

⇒ see the course website for details
Assessment

1. class participation (20%)
2. in class presentation of a topic/paper from the reading list (optional, up to 30%)
3. project (50-80%)

NB: I’ll discuss assignments/presentations/projects later in class…
⇒ but look at the reading list on the web site
DATABASE IMPLEMENTATION

(STANDARD APPROACHES AND 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:
  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
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!)
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, 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).
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 6. Detailed instruction count breakdown for New Order transaction.**
Where does the Time go?

Figure 8. Instructions (left) vs. Cycles (right) for New Order.
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
Compilation-based Approaches

IDEA:

- 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>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>

$$\Sigma_L \quad S_L \quad \varphi \quad \downarrow$$

Logical Schema and User Queries
Compilation-based Approaches

Definability and Rewriting

Queries: range-restricted FOL over $S_L$ definable w.r.t. $\Sigma$ and $S_A$

Schema: range-restricted FOL $\Sigma := \Sigma^L \cup \Sigma^{LP} \cup \Sigma^P$

Data: CWA (complete information for $S_A$ symbols)

Compilation-based Approaches

Definability and Rewriting

Queries range-restricted FOL over $S_L$ definable w.r.t. $\Sigma$ and $S_A$

Schema range-restricted FOL $\Sigma := \Sigma^L \cup \Sigma^{LP} \cup \Sigma^P$

Data CWA (complete information for $S_A$ symbols)

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

Definability and Rewriting

Queries: range-restricted FOL \( \text{over } S_L \) definable

Schema: range-restricted FOL \( \Sigma := \Sigma^L \cup \Sigma^P \)

Data: CWA (complete information for \( S_A \) spec)

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

\[
\begin{align*}
\text{Query (} S_L \text{)} & \xrightarrow{\psi} \text{Compiler} \\
\text{Schema (} S_L \cup S_P \text{)} & \xrightarrow{\Sigma} \text{Evaluator} \\
\text{Data (} S_A \subseteq S_P \text{)} & \xrightarrow{(\text{instance of) } S_A} \text{Answers}
\end{align*}
\]
% ------- conceptual modelling -------
% disjoint coverage
student(x,y) <-> (ugrad(x,y) or grad(x,y)),
ugrad(x,y) and grad(x,z) -> bot,
% student id is a key
student(x,y) and student(x,z) -> y=z,
%
% ------- physical modelling -------
% two access paths: p0astudent and p1agrad use record ids
student(x,y) <-> ex(r,p0astudent(r,x,y)),
grad(x,y) <-> ex(r,p0astudent(r,x,y) and p1agrad(r)),
% record ids are keys too
p0astudent(r,x,y) and p0astudent(r,z,w) -> x=z,
p0astudent(r,x,y) and p0astudent(s,x,z) -> r=s,
%
% ------- queries -------
q0gs(x,y) <-> grad(x,y),
q0us(x,y) <-> ugrad(x,y)
Example (cont.)

david@david$ cat tests/old_format/848ex/students.fol | ...

query(q0gs,2,0,[var(1,int),var(2,int)]) <->
project([var(3,int)],

nlj(
   ap(p0astudent,[var(3,int),var(1,int),var(2,int)],fscan)
   ap(p1agrad,[var(3,int)],flookup,1)
)
)

query(q0us,2,0,[var(1,int),var(2,int)]) <->
project([var(3,int)],

nlj(
   ap(p0astudent,[var(3,int),var(1,int),var(2,int)],fscan)
   complement(
       ap(p1agrad,[var(3,int)],flookup,1)
   )
)
)
#include "runtime.h"
// struct for us
struct us_data {
  // public:
  long   var_1;
  long   var_2;
  long   var_3;
  
  // operators:

  // AP private:
  struct fscan_data apvar0;
  struct flookup_data apvar1;
};
Example (C source: `us.c`)

```c
#include <stdio.h>
#include <stdlib.h>
#include "us.h"

static int inline __attribute__((always_inline)) getfirst_simpcomp2(struct us_data *q) {
    if (getfirst_p1agrad(&(q->apvar1), &(q->var_3))) return 0;
    return 1;
};
static int inline __attribute__((always_inline)) getnext_simpcomp2(struct us_data *q) {
    return 0;
};

static int inline __attribute__((always_inline)) getfirst_nlj3(struct us_data *q) {
    if (!getfirst_p0astudent(&(q->apvar0), &(q->var_3), &(q->var_1), &(q->var_2))) return 0;
    while (!getfirst_simpcomp2(q))
        if (!getnext_p0astudent(&(q->apvar0), &(q->var_3), &(q->var_1), &(q->var_2))) return 0;
    return 1;
};
static int inline __attribute__((always_inline)) getnext_nlj3(struct us_data *q) {
    if (getnext_simpcomp2(q)) return 1;
    while (getnext_p0astudent(&(q->apvar0), &(q->var_3), &(q->var_1), &(q->var_2)))
        if (getfirst_simpcomp2(q)) return 1;
    return 0;
};

static int inline __attribute__((always_inline)) getfirst_project4(struct us_data *q) {
    return getfirst_nlj3(q);
};
static int inline __attribute__((always_inline)) getnext_project4(struct us_data *q) {
    return getnext_nlj3(q);
};
```
Take Home

Focus of this class: DB engine vs. Compilation approaches

Lots of open issues:

1. Main memory data organization
2. Multilevel memory/storage
3. Ordered data
4. Parallelism and partitioning (many levels)
5. ...

Next time:

1. Basics of DB implementation (crash course)
2. Basics of Query Compilation (with examples)
3. Discussion of presentations/projects
Take Home

Focus of this class: DB engine vs. Compilation aproaches

Lots of open issues:
1. Main memory data organization
2. Multilevel memory/storage
3. Ordered data
4. Parallelism and partitioning (many levels)
5. ...

Next time
1. Basics of DB implementation (crash course)
2. Basics of Query Compilation (with examples)
3. Discussion of presentations/projects
Take Home

Focus of this class: DB engine vs. Compilation approaches

Lots of open issues:

1. Main memory data organization
2. Multilevel memory/storage
3. Ordered data
4. Parallelism and partitioning (many levels)
5. . .

Next time

1. Basics of DB implementation (crash course)
2. Basics of Query Compilation (with examples)
3. Discussion of presentations/projects