HyPer
A Hybrid OLTP & OLAP Main Memory Database System

Alfons Kemper, Thomas Neumann
Presented By: Brad Glasbergen
Database Workloads

OLTP
- Fast
- Constrained
- Write-heavy

OLAP
- Long
- Complex
- Read-heavy
## OLTP Database Design

<table>
<thead>
<tr>
<th>ORDER</th>
<th>DISTRICT</th>
<th>WAREHOUSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
### OLTP Database Design

<table>
<thead>
<tr>
<th>ORDER</th>
<th>DISTRICT</th>
<th>WAREHOUSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
OLTP Database Design

<table>
<thead>
<tr>
<th>ORDER</th>
<th>DISTRICT</th>
<th>WAREHOUSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

1
1
1
## OLTP Database Design

<table>
<thead>
<tr>
<th>ORDER</th>
<th>DISTRICT</th>
<th>WAREHOUSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

The table above shows the relationship between orders, districts, and warehouses. Each order is associated with a specific district and warehouse.
### OLTP Database Design

<table>
<thead>
<tr>
<th>ORDER</th>
<th>DISTRICT</th>
<th>WAREHOUSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

**Insert Statement:**

```
INSERT ORDER INTO NEW_ORDERS ... 
```
### OLTP Database Design

<table>
<thead>
<tr>
<th>ORDER</th>
<th>DISTRICT</th>
<th>WAREHOUSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

**INSERT ORDER INTO NEW_ORDERS ...**
# OLTP Database Design

**Query:**
```
SELECT COUNT(*) …
GROUP BY WAREHOUSE
```
### OLTP Database Design

**SELECT COUNT(*) ... GROUP BY WAREHOUSE**

<table>
<thead>
<tr>
<th>ORDER</th>
<th>DISTRICT</th>
<th>WAREHOUSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

![Diagram of warehouse distribution](image-url)
OLTP Database Design

SELECT COUNT(*) ...
GROUP BY WAREHOUSE

Long-Held Conflicting Locks
## OLAP Database Design

<table>
<thead>
<tr>
<th>ORDER</th>
<th>DISTRICT</th>
<th>WAREHOUSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
## OLAP Database Design

<table>
<thead>
<tr>
<th>ORDER</th>
<th>DISTRICT</th>
<th>WAREHOUSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

1

2
OLAP Database Design

<table>
<thead>
<tr>
<th>ORDER</th>
<th>DISTRICT</th>
<th>WAREHOUSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

SELECT COUNT(*) ... GROUP BY WAREHOUSE
### OLAP Database Design

#### SELECT COUNT(*) ...
GROUP BY WAREHOUSE
INSERT INTO ORDERS ...
The State of the Art

OLTP

Load

Delay

OLAP
1.5 TB DRAM ~ 50k

Put Everything In Memory
In-Memory OLTP Engines

1 1 1
1 1 1
1 1 1
2 2 2
2 2 2
3 3 3
3 3 3
3 3 3
1 1 1
1 1 1
1 1 1
2 2 2
2 2 2
3 3 3
3 3 3
3 3 3
Partitioned OLTP Transactions

Op Queue

1
1
1

Order  Order

2
2
2

Order

3
3
3

Order  Order
Bad at Analytics Transactions!
HyPer: Fork a Snapshot!

Partition 1

Partition 2

Dedicated OLTP threads

Dedicated OLAP process
HyPer: Fork a Snapshot!

Partition 1  Order

Partition 2  Order

Fork
HyPer: Fork a Snapshot!

Partition 1

Order  Order  Order  Order  Order

Partition 2

Order  Order  Order  Order  Order

Fork  Analytics
Fork: Technical Details

Page 1
Page 2
Page 3
...
Page N

Fork()
Fork: Technical Details

Page 1

Page 2

Page 3

...

Page N

Fork()
Fork: Copy on Write

Page 1
Page 2
Page 3
...
Page N

Copy on Write
Fork: Copy on Write

- Page 1
- Page 2
- Page 3
- ...
- Page N

Page 1
Page 3
Fork: Freeing Memory
Cleaning Dirty Snapshots

Partition 1

Order | Order | Order | Order | Order | Order

Partition 2

Order | Order | Order | Order | Order | Order

Fork | Analytics

In-flight data, inconsistent!
Cleaning Dirty Snapshots

Partition 1

Undo Logs

Order

Fork

Undo

Analytics

Persist

Undo
Logging Bottlenecks

Order

Order
Logging Bottlenecks

Time

Run C

Run
Logging Bottlenecks

Time

Run C

Run C
Logging Bottlenecks

```
Run  C
```

```
Run  C
```
Logging Bottlenecks

Time

Run C

Commit 1
Logging Bottlenecks

Time

Commit 1, Commit 2

Run

C

Run

C
Logging Bottlenecks

As Time progresses, the system Run increases, which leads to an increase in Latency and an increase in Throughput. The ACK signal is sent to the system to acknowledge the completion of the operation.

Increases Latency, Increases Throughput
Evaluation

OLTP

TPC-C Order Entry Benchmark

OLAP

TPC-H Analysis Benchmark
OLTP Throughput

<table>
<thead>
<tr>
<th></th>
<th>HyPer (single thread)</th>
<th>VoltDB (single node)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Throughput</td>
<td>126 K</td>
<td>55 K</td>
</tr>
</tbody>
</table>

Why?
OLTP/OLAP Throughput

<table>
<thead>
<tr>
<th></th>
<th>OLTP</th>
<th>OLAP</th>
</tr>
</thead>
<tbody>
<tr>
<td>OLTP</td>
<td>5x</td>
<td></td>
</tr>
<tr>
<td>OLAP</td>
<td></td>
<td>3x</td>
</tr>
</tbody>
</table>

HyPer

- 380 K

VoltDB

- 300 K

Competitive with MonetDB
Leverage OS fork() to make efficient snapshots!
Efficiently Compiling Efficient Query Plans for Modern Hardware

Thomas Neumann
Technische Universität München
neumann@in.tum.de

Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems

Thomas Neumann
Technische Universität München
neumann@in.tum.de

Tobias Möhlbauer
neumann@martin-reiter.net

Alfons Kemper
Alfons.Kemper@in.tum.de

ABSTRACT

As database systems, query performance is more and more determined by the raw CPU cost of processing queries itself. The classical database query-processing techniques, such as the use of columnar stored tables, are less efficient when considering modern architectures. CPUs due to their lack of hardware and frequent instruction-based parallelism. Several techniques like hash-based partitioning or re-indexing of tables have been proposed in the past to improve this situation, but even those techniques are frequently outperformed by hand-written execution plans.

Our approach improves the situation by employing a compiler strategy that translates a query into compact and efficient machine code using the LLVM compiler framework. By analyzing and optimizing code and data locality and predictable branch latency of the resulting code, we improve the performance of modern CPUs.

1. INTRODUCTION

This work proposes a compiler strategy to ensure fast compilation of SQL queries into efficient machine code. This is achieved by transforming query execution plans into a form that can be compiled into native code.

Fast Serializable Multi-Version Concurrency Control

Thomas Neumann
Technische Universität München
neumann@in.tum.de

Tobias Möhlbauer
neumann@martin-reiter.net

Alfons Kemper
Alfons.Kemper@in.tum.de

ABSTRACT

Multi-Version Concurrency Control (MVCC) is a widely used concurrency control mechanism, as it allows for consistent views even when readers may change a database. However, most systems implement only explicit snapshot isolation (SI) instead of MVCC. Although the presence of MVCC has been shown to improve performance in some scenarios, the practicality of MVCC remains to be proven.

Our approach enables a database system to support both MVCC and SI in a single transaction, thereby providing users with the benefits of both isolation levels. By implementing MVCC, we achieve performance improvements over SI alone, as it allows for more efficient query evaluation. Additionally, MVCC provides stronger isolation guarantees, ensuring that transactions do not interfere with each other in a way that SI cannot.

Categories and Subject Descriptors

• D.2.4 Database Management Systems

Keywords

• D.2.4.7 Transactions

1. INTRODUCTION

Transaction isolation is one of the most fundamental issues addressed by database management systems (DBMS). It provides a level of protection that allows concurrent transactions to access data concurrently without interfering with each other.

In the background, the DBMS ensures that the resulting transactions are consistent and executes all changes atomically. This is achieved by implementing a series of isolation levels, which define how transactions are isolated from each other.

In this paper, we present a novel approach to implement MVCC in a way that is both efficient and scalable. By using our implementation, we achieve significant performance improvements over existing SI systems, while still providing strong isolation guarantees.
RUMA has It: Rewired User-space Memory Access Is Possible!

Feike Martin Schuhkinschat, Jens Dittrich, Ankur Sharm

Information Systems Group
info-sys.cs.uni-aarland.de

Abstract
Memory management is one of the many active areas in database research. It plays a crucial role in tasks like free-space management on solid-state storage media. Here, and elsewhere, we also realize its impact on database performance. When worrying about NVM-aware memory allocation, data compaction, garbage collection, and other generation, most often, it's not to the same scope of anomaly as existing "archival solutions" or "locking across memory banks".

What if there were techniques that would provide memory management from a third helper things to first class citizen in algorithm and systems design? What if but technique turned the role of memory management into a classhers open and any other data processing system specific tools? What if this technique could be identified as a key here designing various core algorithms with the effect of supercharging existing state-of-the-art methods and applications? Then we would write this paper.

We introduce RUMA, Rewired User-space Memory Access. It allows for physiological design management, i.e., we develop algorithms to deeply analyze the mapping from virtual to physical memory in user-space world, at the same time enabling the virtual memory support of highly hardware and operating system. We show that fundamental data structures like maps to array, partitioning, sorting, and parallelization become strong from RUMA.

1. INTRODUCTION

In a multi-management systems handle memory at multiple layers in various formats. The allocation differs in size, i.e., request memory, and lifetime. Many programs need a memory management as a necessary evil that is completely decoupled from their application code's schema design. They claim and release memory using a technique that is not type efficient (when), without considering the state of their allocation patterns in the system.

This control of allocation can strike back on itself. A generic example to counter this behavior is named pooling. With classical allocation, (e.g., malloc) it is unclear whether an allocation is served from pooled memory or via the allocation of fresh pages requested from the kernel. The difference between them is significant. Requesting free pages from the system is extremely expensive as a program must be interrupted and the kernel has to initialize the new page. But, before the program can continue the execution.

Thus, (most) engineers implement their own pooling system to get full control over the memory allocation and to reuse portion of it as if it was as possible. However, manual pooling also complicates things. To write a script to programs, engineers rely on extensive memory regions. Any programs access to these data in large anonymous arrays. Data structures on that data as are space expensive to reside in memory locations. This model is unhandled deeply in state-of-the-art systems. For instance, the authors of [1] that accessing the request to in a dual-operating system on sub-storage locations for more D:

"It does not allow for eighty-eight-page leaps without further method (and a test) that it is not the most of the dual-operating system"

Unfortunately, it is not always possible to gather large consecutively memory regions from the pool due to fragmentation. To


This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/ For any use beyond those covered by this license, additional permission may be required.

"Rewiring the Memory": 2016 Edition, Vol. 8, No. 30
Copyright 2016 VLDB Endowment 259-9037/16/00.

Figure 1: Structural Synergies in Design Performance

Picture: While showing the synergy between providing awareness and memory access performance, this figure highlights how different systems (e.g., databases, operating systems, etc.) can work together to optimize the look up of services in a direct and close relationship. Rewired memory offers a

"Drawback Parallelism"

"Synergetic Structures" (e.g., Arrays)
Discussion Questions:

- Are there other operating systems primitives that we can leverage in databases?
- How “real-time” do real-time analytics *need* to be?
- How do we scale-out HyPer’s OLTP engine? Do we need to?
- Update propagation to secondary servers