HyPer

A Hybrid OLTP & OLAP Main Memory Database System

Alfons Kemper, Thomas Neumann
Presented By: Brad Glasbergen
UNIVERSITY OF WATERLOO

Database Workloads

OLTP

- Fast
- Constrained
- Write-heavy

OLAP

- Long
- Complex
- Read-heavy

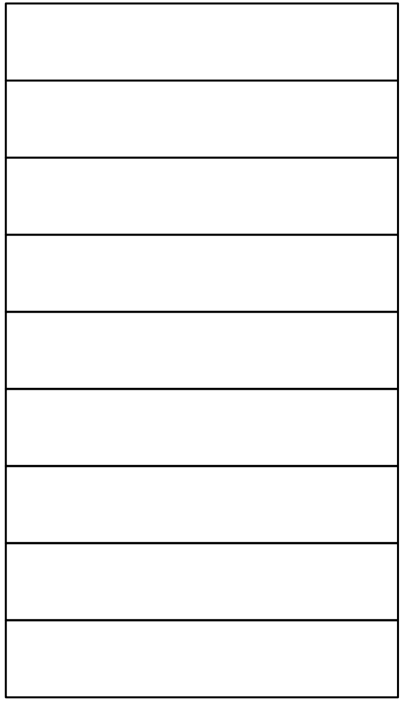
OLTP Database Design

ORDER	DISTRICT	WAREHOUSE
1	1	1
2	2	2

UNIVERSITY OF WATERLOO

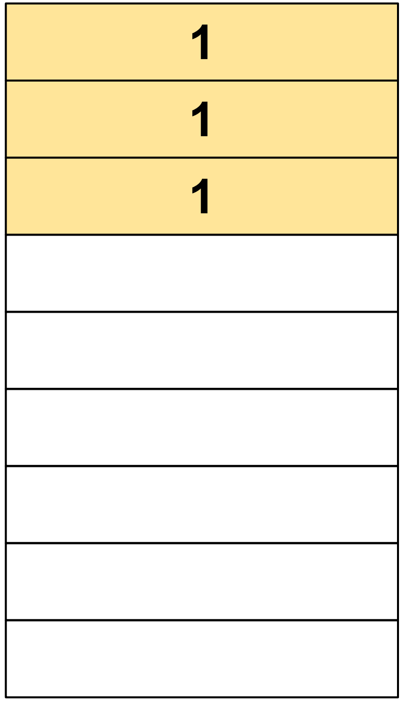
OLTP Database Design

ORDER	DISTRICT	WAREHOUSE
1	1	1
2	2	2



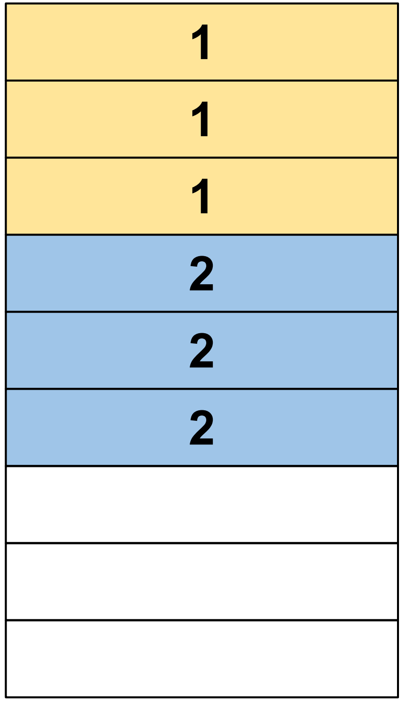
OLTP Database Design

ORDER	DISTRICT	WAREHOUSE
1	1	1
2	2	2



OLTP Database Design

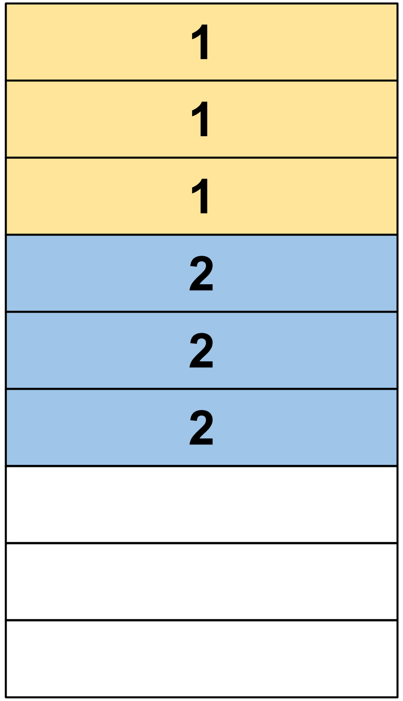
ORDER	DISTRICT	WAREHOUSE
1	1	1
2	2	2



OLTP Database Design

ORDER	DISTRICT	WAREHOUSE
1	1	1
2	2	2

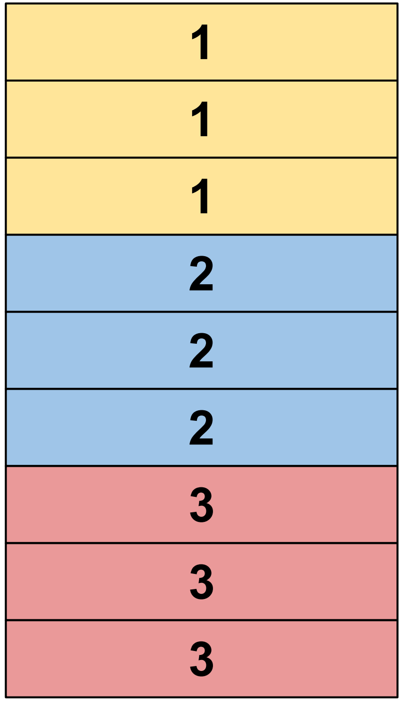
INSERT ORDER INTO NEW_ORDERS …



OLTP Database Design

ORDER	DISTRICT	WAREHOUSE
1	1	1
2	2	2
3	3	3

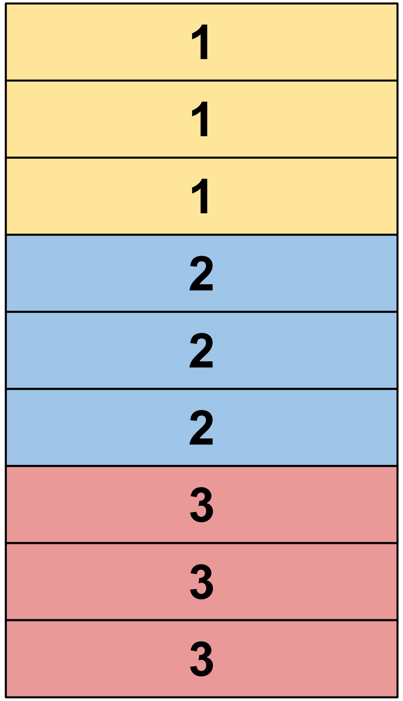
INSERT ORDER INTO NEW_ORDERS …



OLTP Database Design

ORDER	DISTRICT	WAREHOUSE
1	1	1
2	2	2
3	3	3

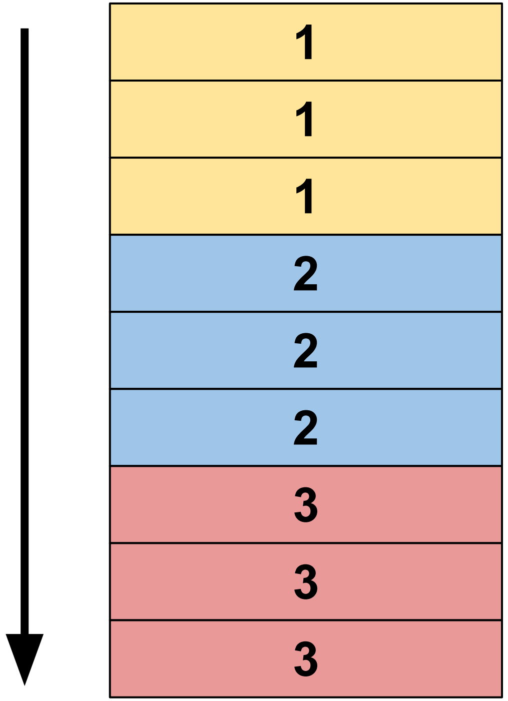
SELECT COUNT(*) …
GROUP BY WAREHOUSE



OLTP Database Design

ORDER	DISTRICT	WAREHOUSE
1	1	1
2	2	2
3	3	3

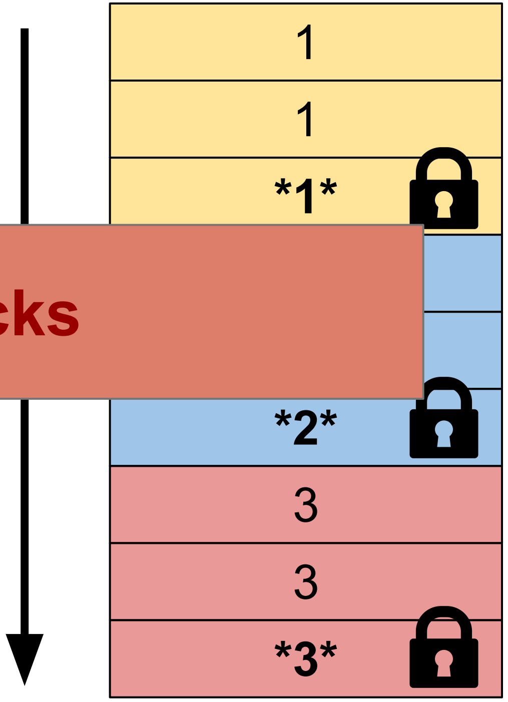
SELECT COUNT(*) …
GROUP BY WAREHOUSE



OLTP Database Design

ORDER	DISTRICT	WAREHOUSE
1		
2	2	2
3	3	3

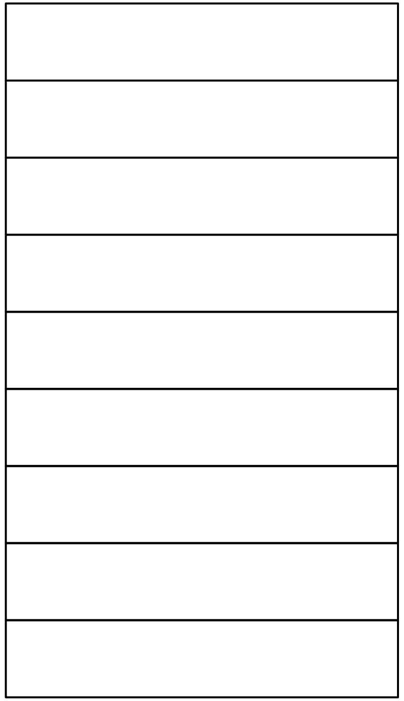
Long-Held Conflicting Locks



SELECT COUNT(*) …
GROUP BY WAREHOUSE

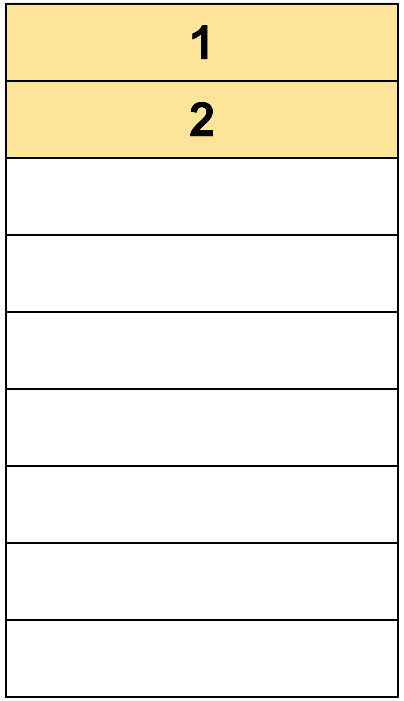
OLAP Database Design

ORDER	DISTRICT	WAREHOUSE
1	1	1
2	2	2



OLAP Database Design

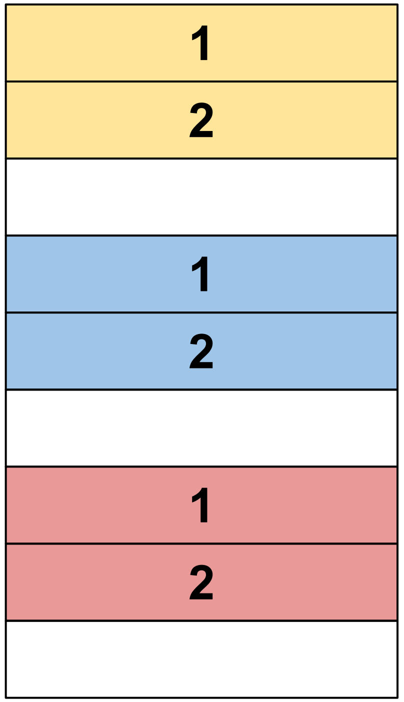
ORDER	DISTRICT	WAREHOUSE
1	1	1
2	2	2



OLAP Database Design

ORDER	DISTRICT	WAREHOUSE
1	1	1
2	2	2

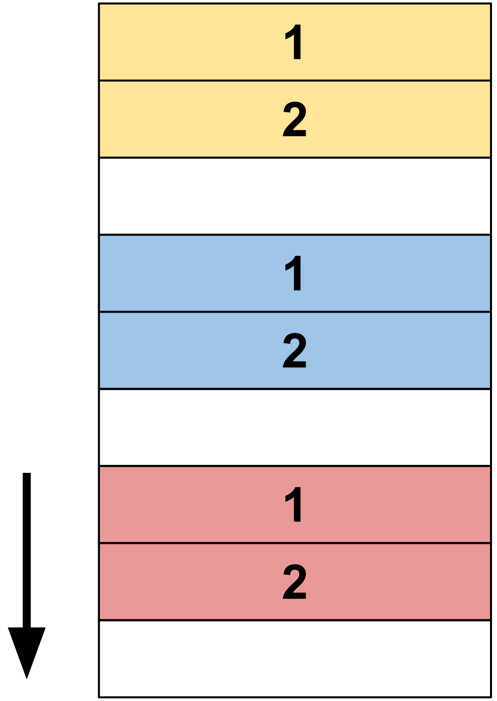
SELECT COUNT(*) …
GROUP BY WAREHOUSE



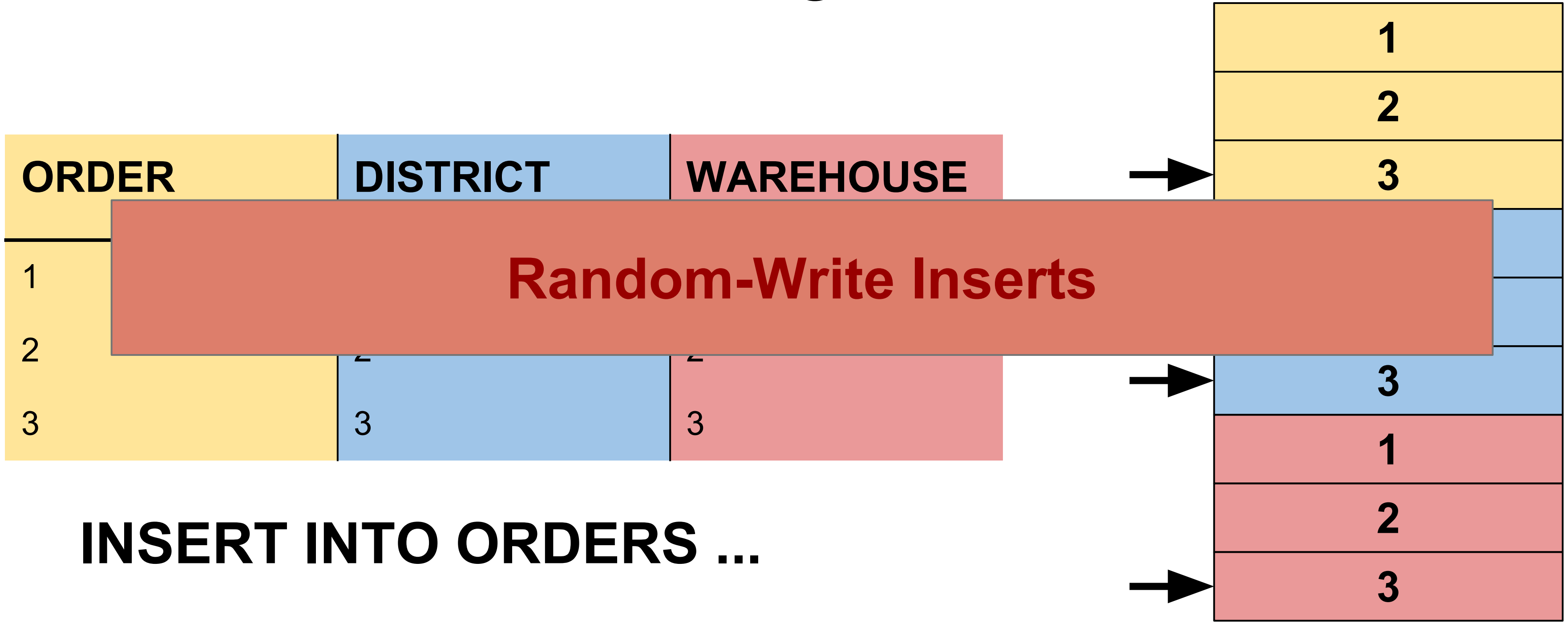
OLAP Database Design

ORDER	DISTRICT	WAREHOUSE
1	1	1
2	2	2

```
SELECT COUNT(*) …
GROUP BY WAREHOUSE
```



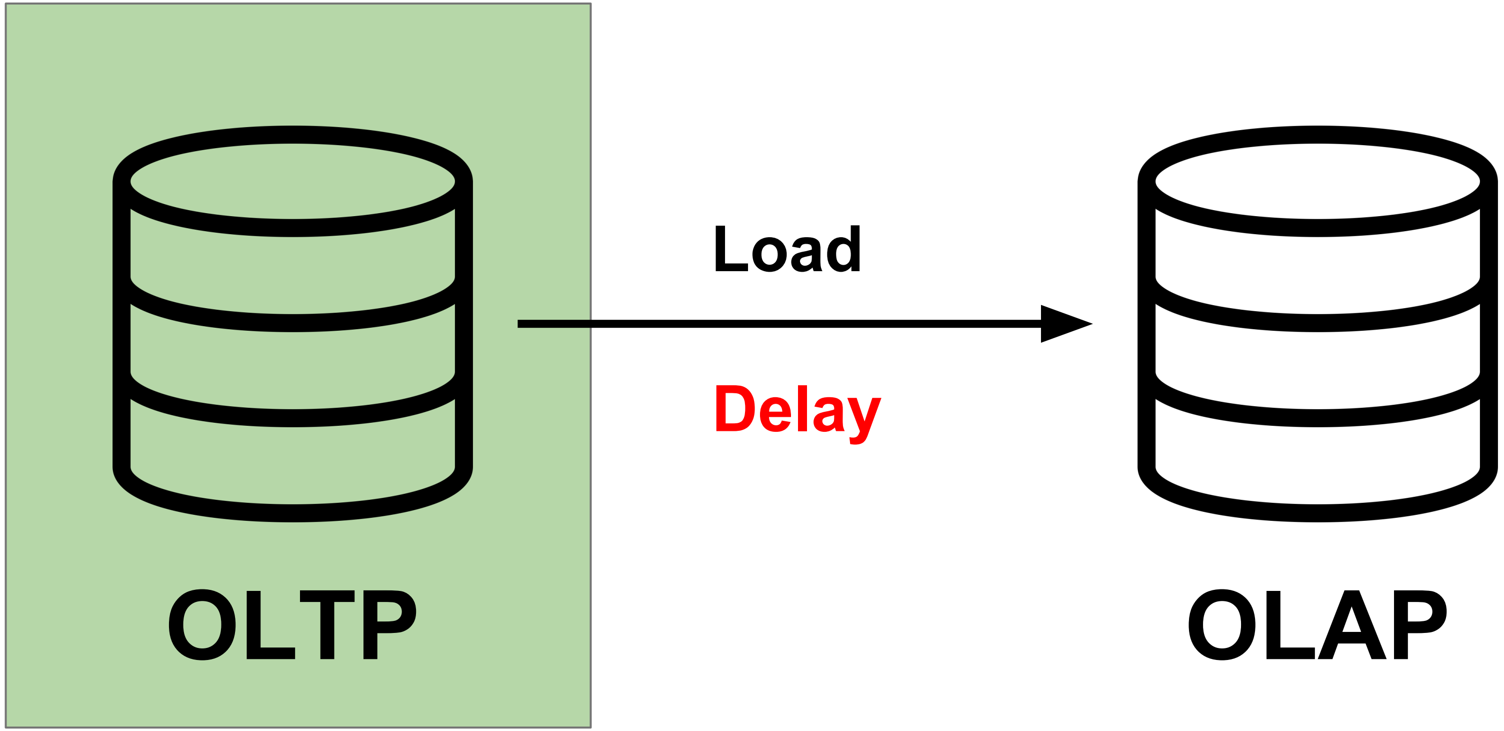
OLAP Database Design



Random-Write Inserts

INSERT INTO ORDERS ...

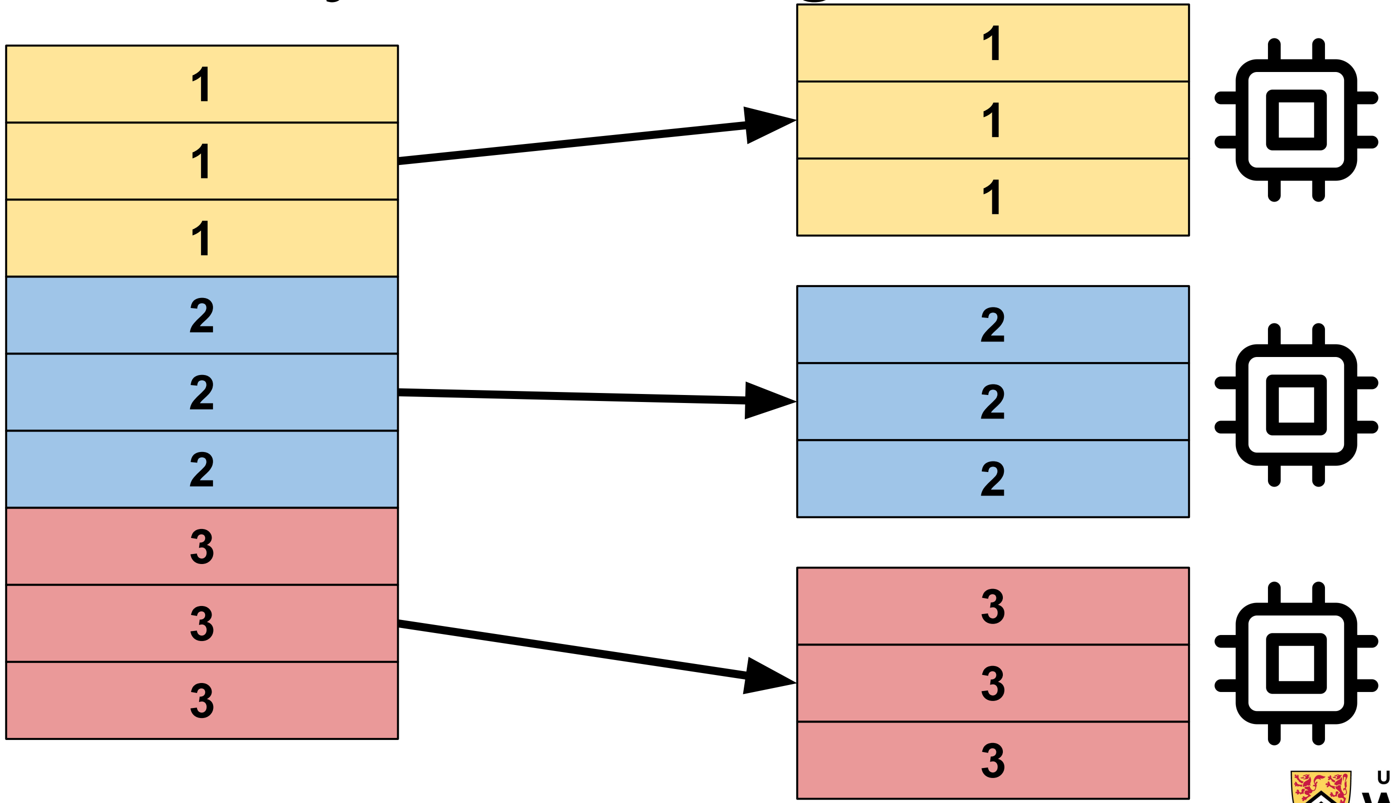
The State of the Art



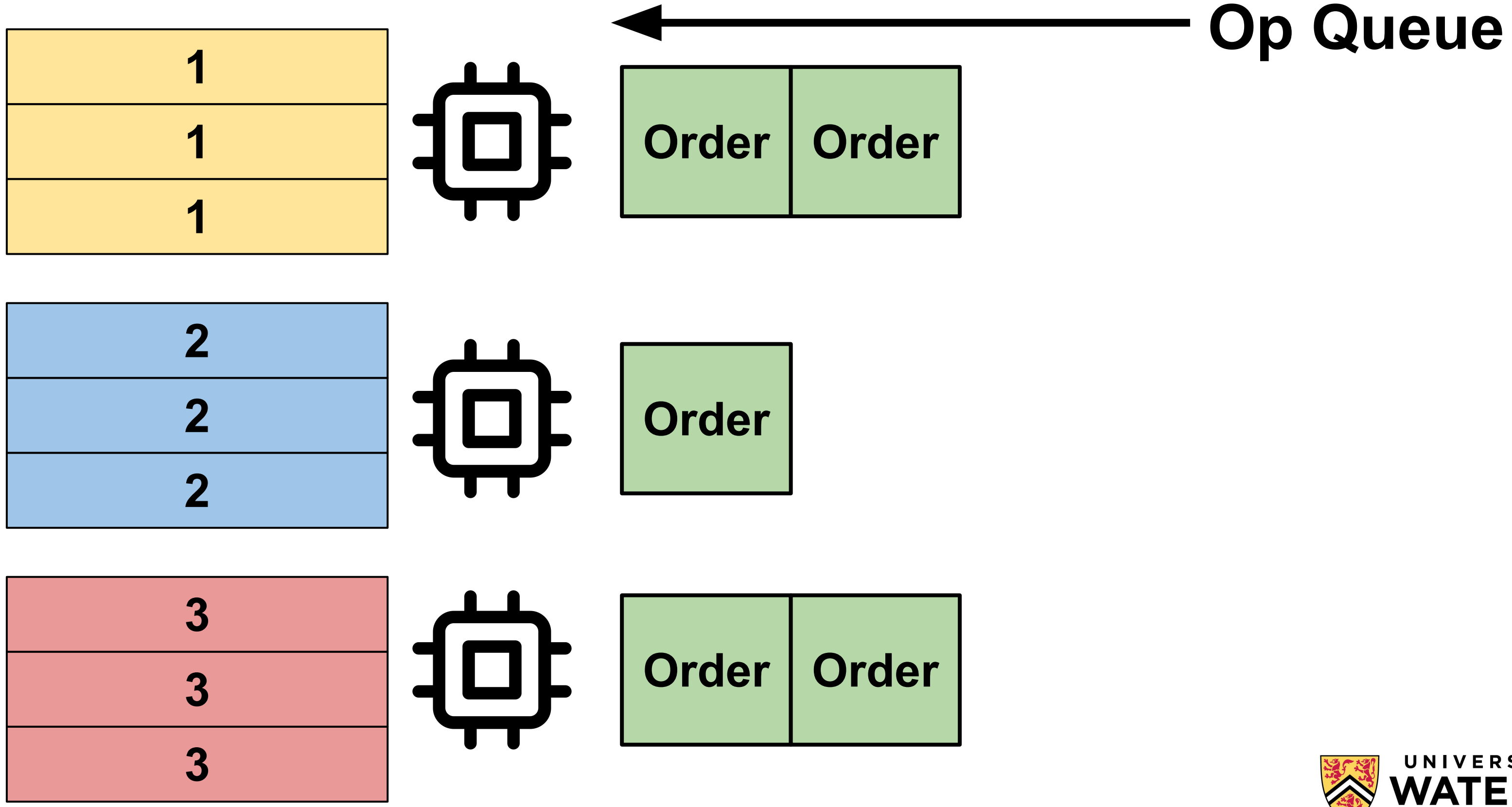
1.5 TB DRAM ~ 50k

Put Everything In Memory

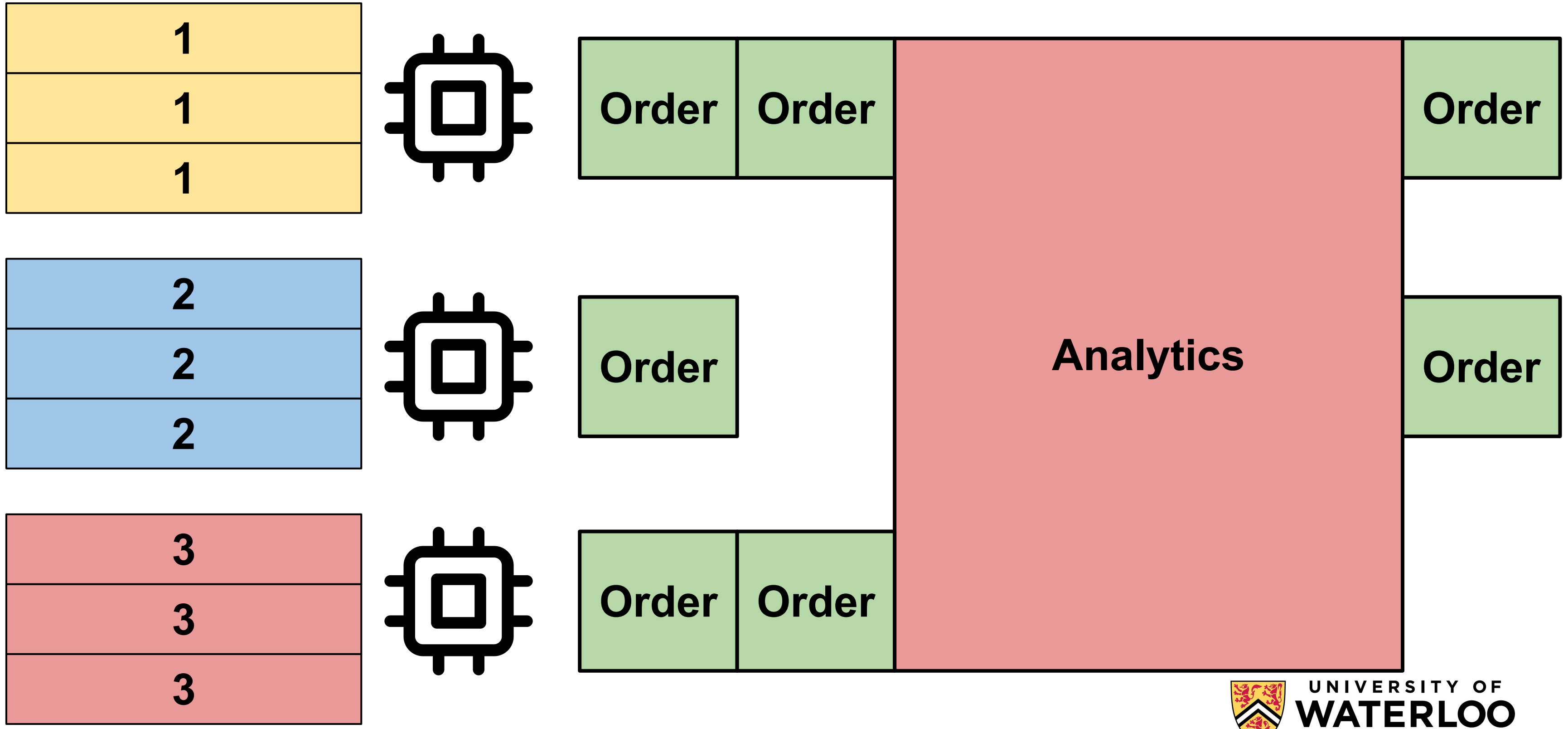
In-Memory OLTP Engines



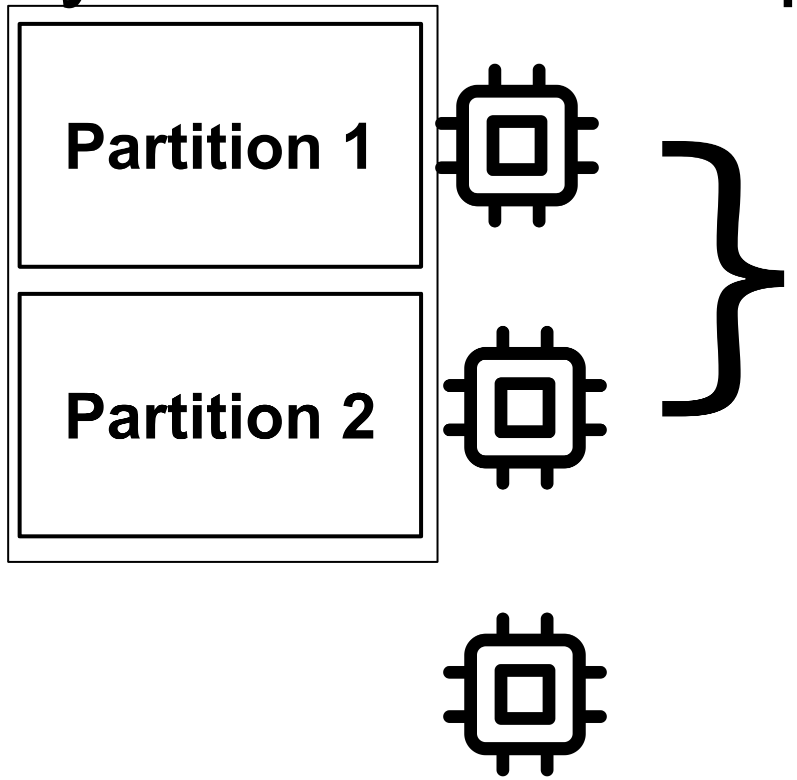
Partitioned OLTP Transactions



Bad at Analytics Transactions!



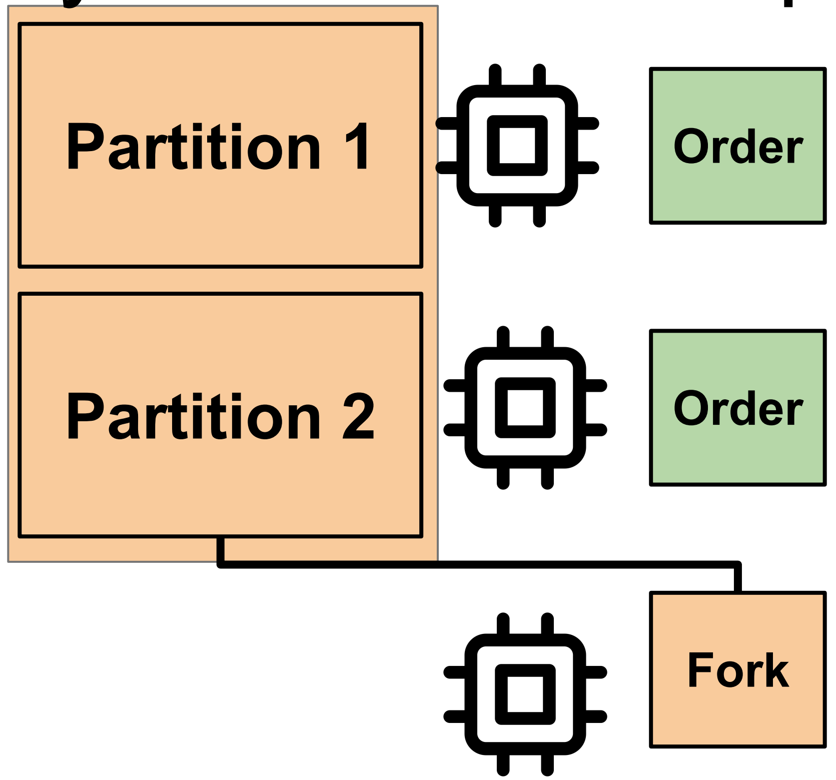
HyPer: Fork a Snapshot!



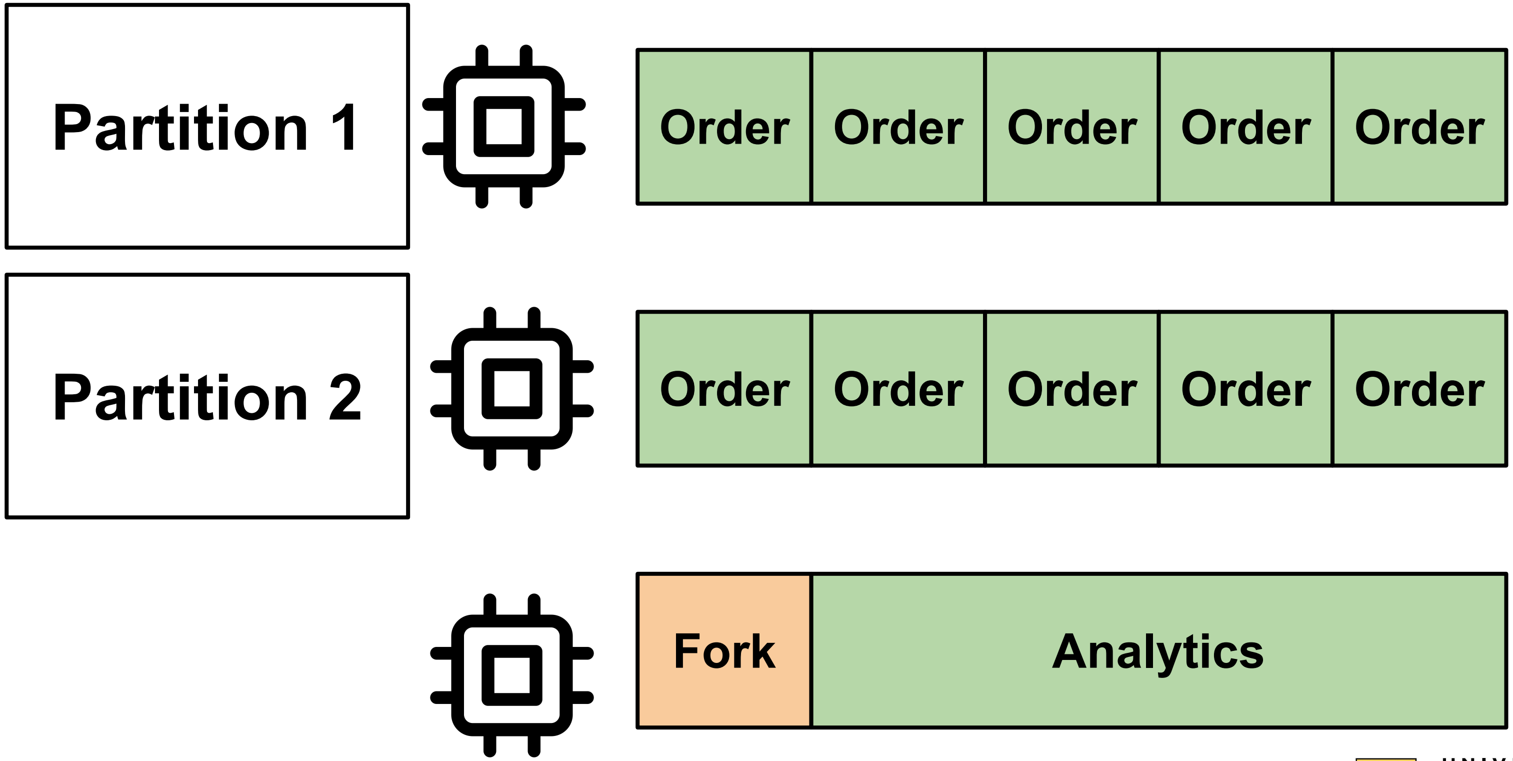
Dedicated OLTP threads

Dedicated OLAP process

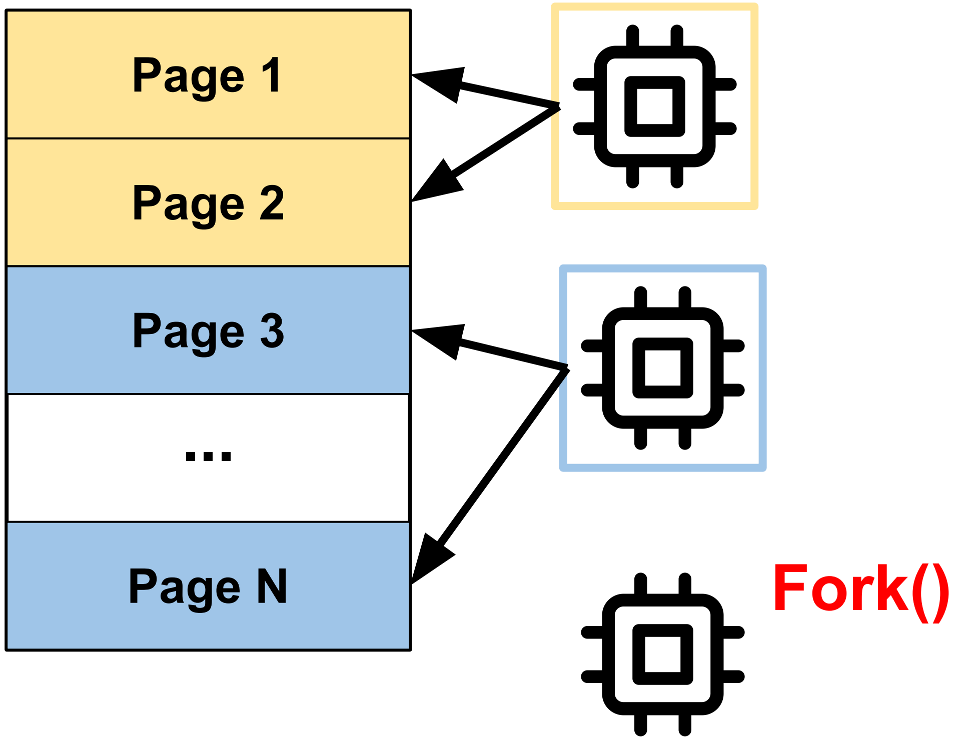
HyPer: Fork a Snapshot!



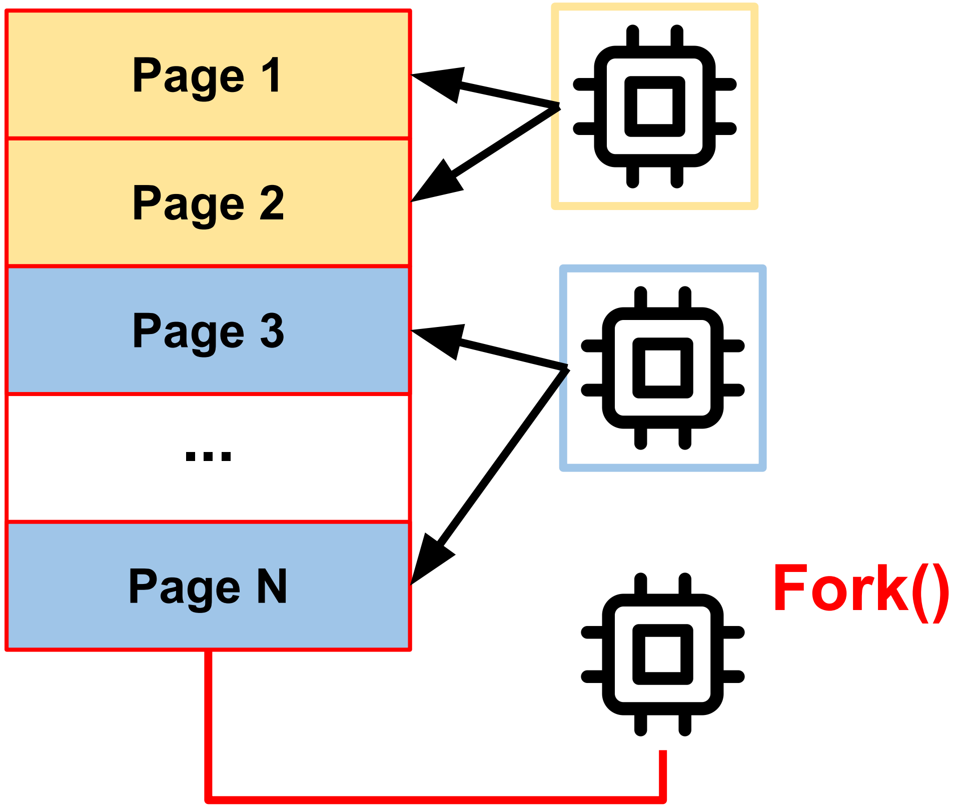
HyPer: Fork a Snapshot!



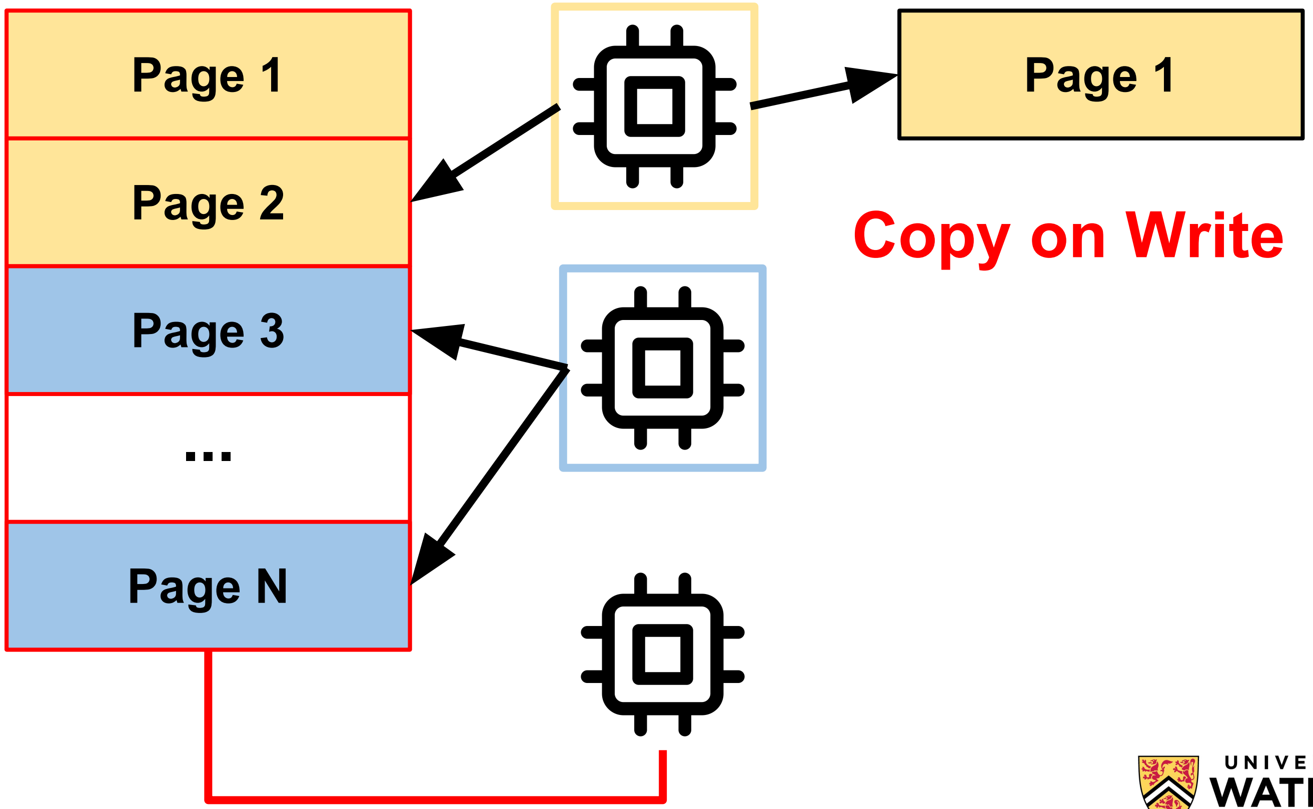
Fork: Technical Details



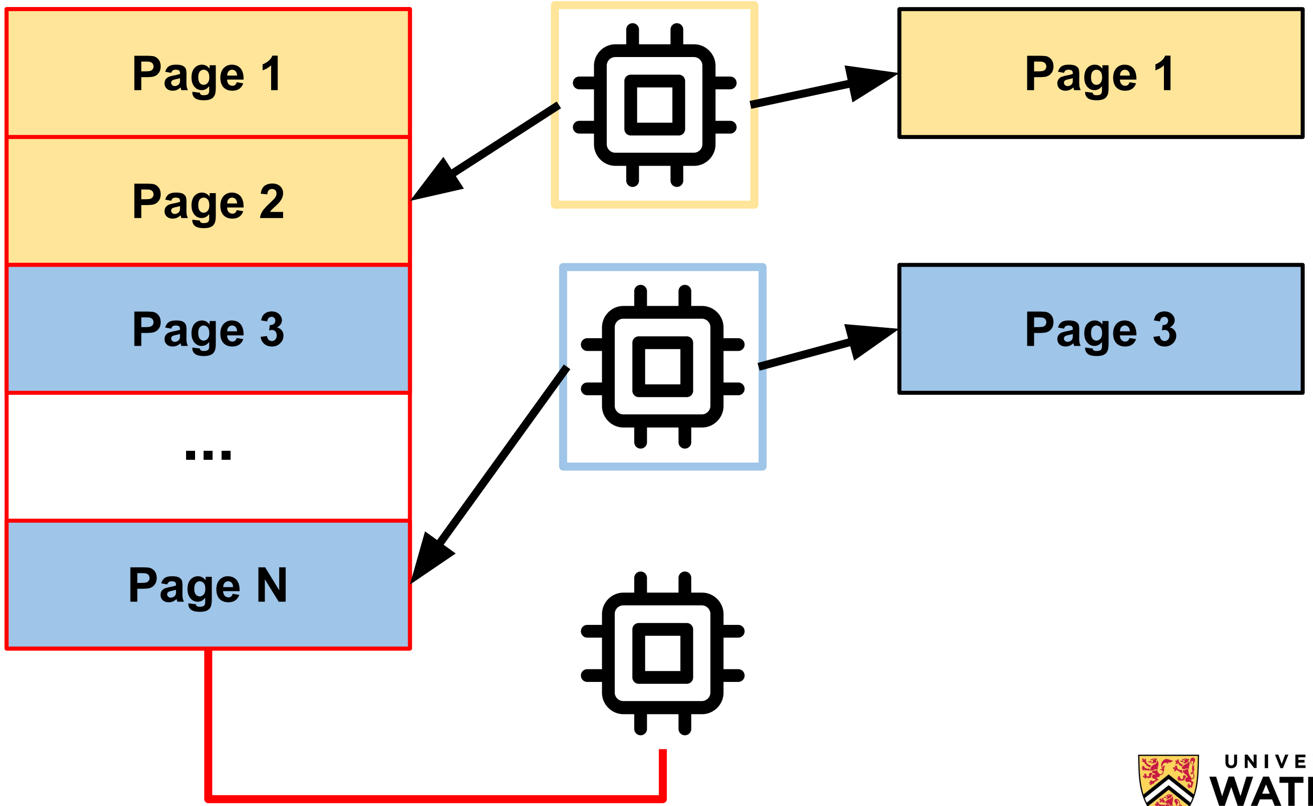
Fork: Technical Details



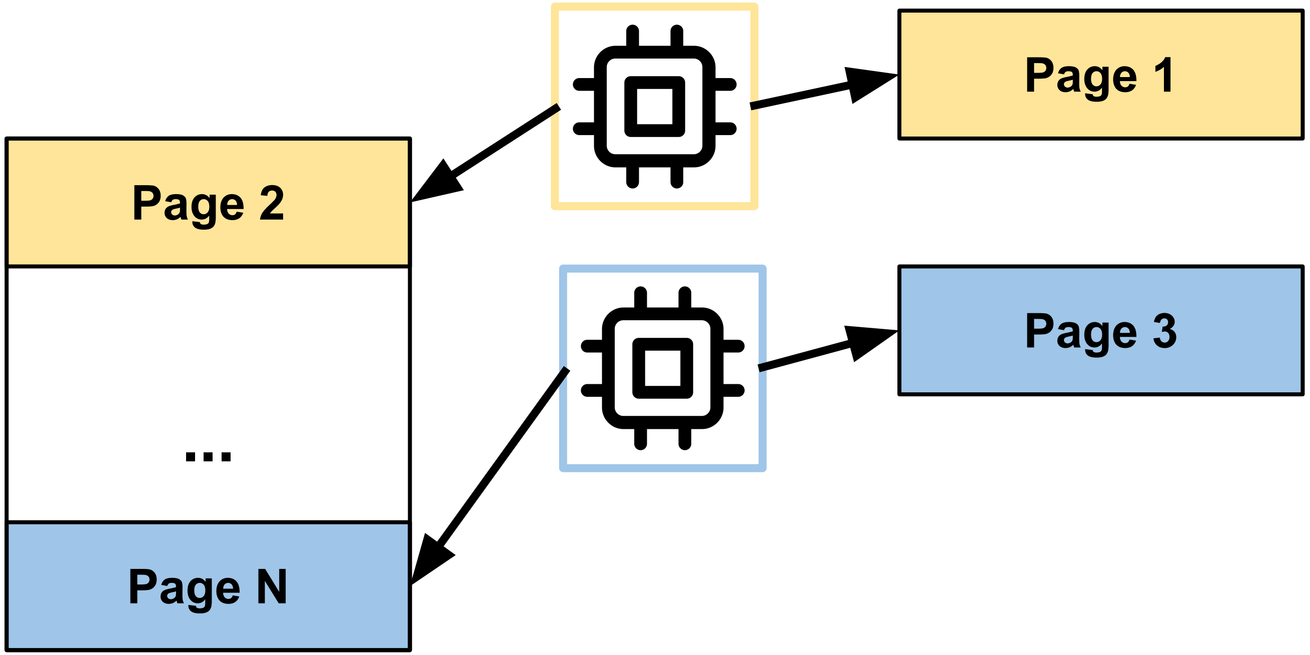
Fork: Copy on Write



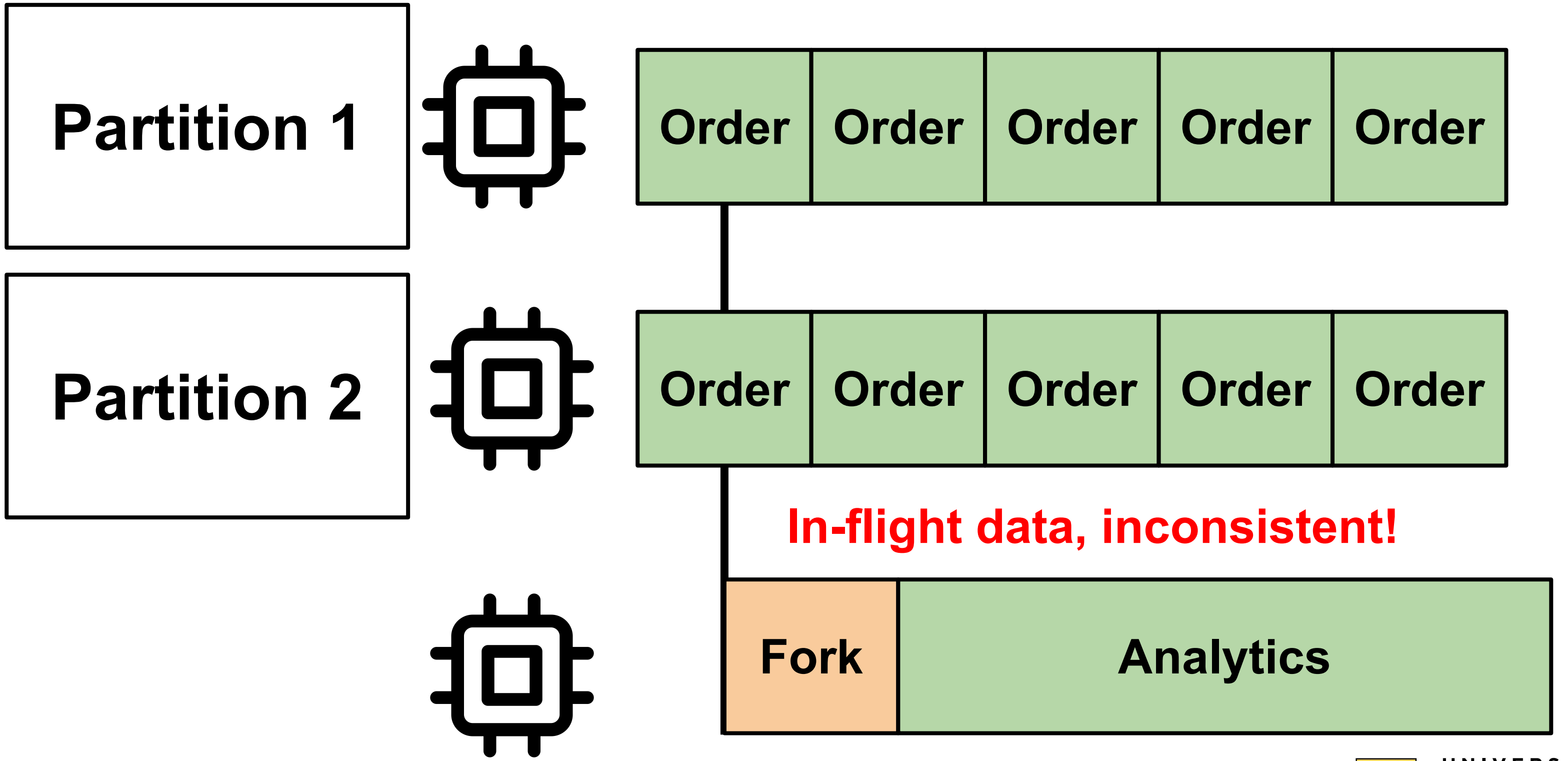
Fork: Copy on Write



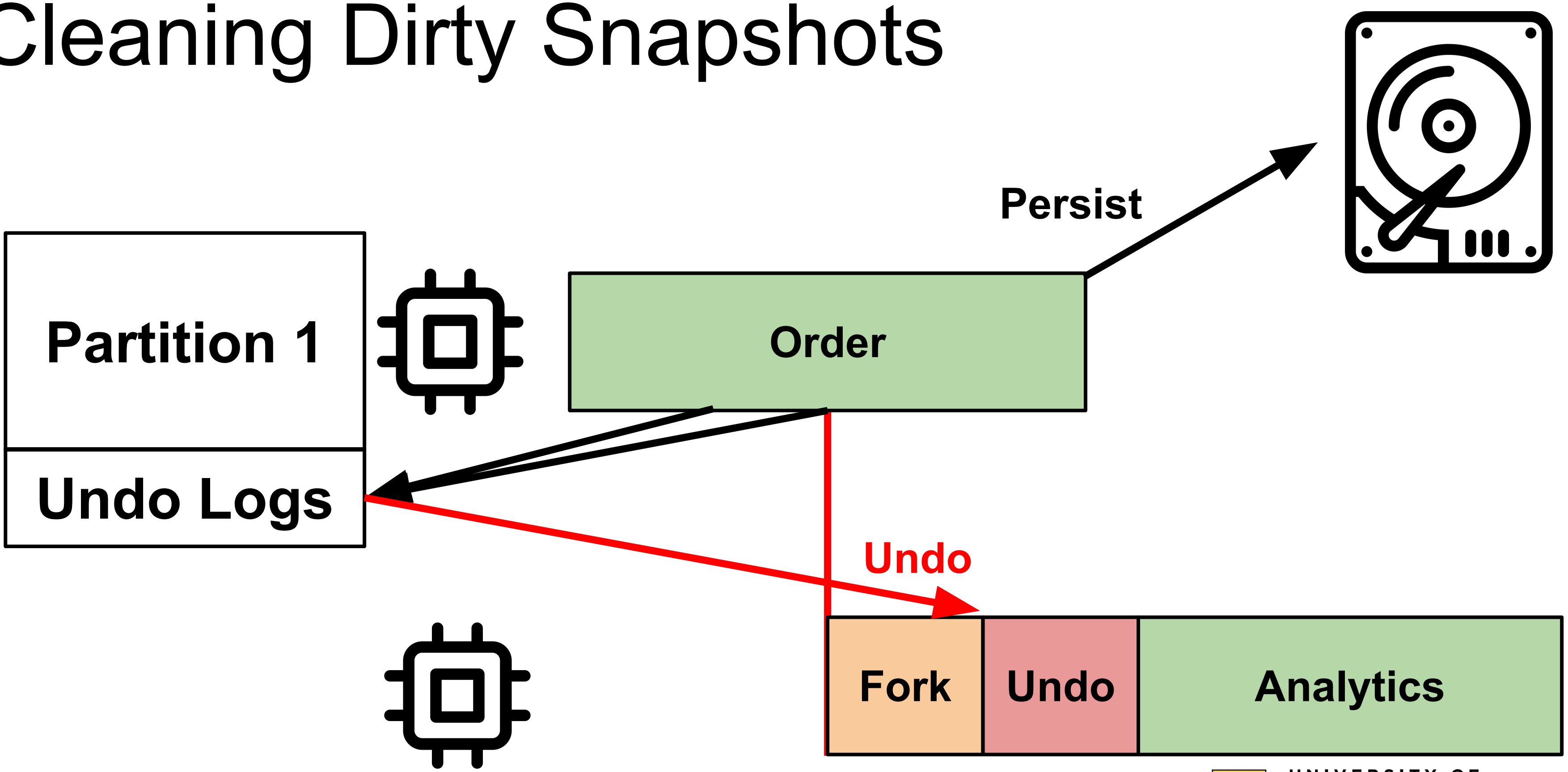
Fork: Freeing Memory



Cleaning Dirty Snapshots

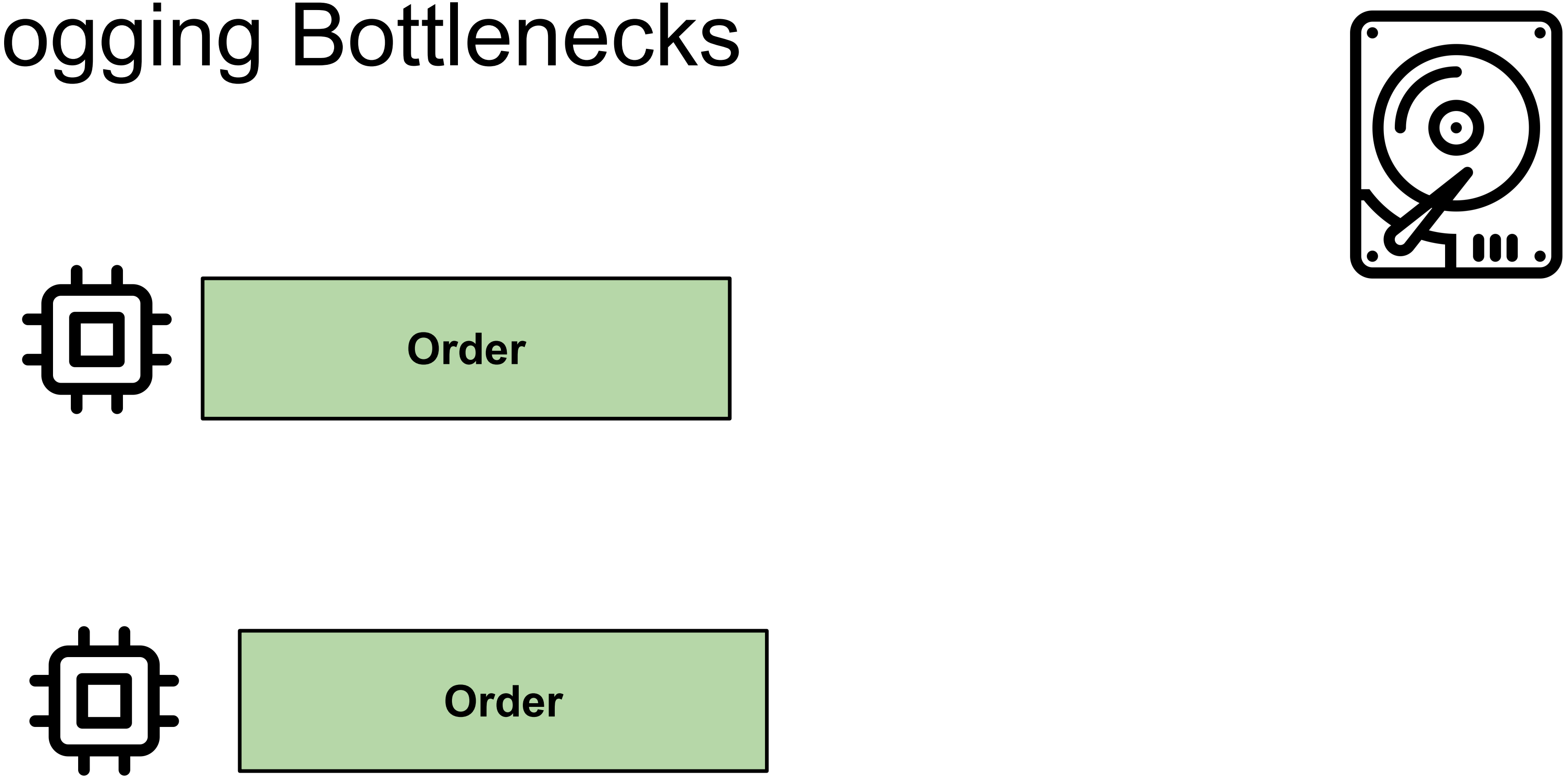


Cleaning Dirty Snapshots

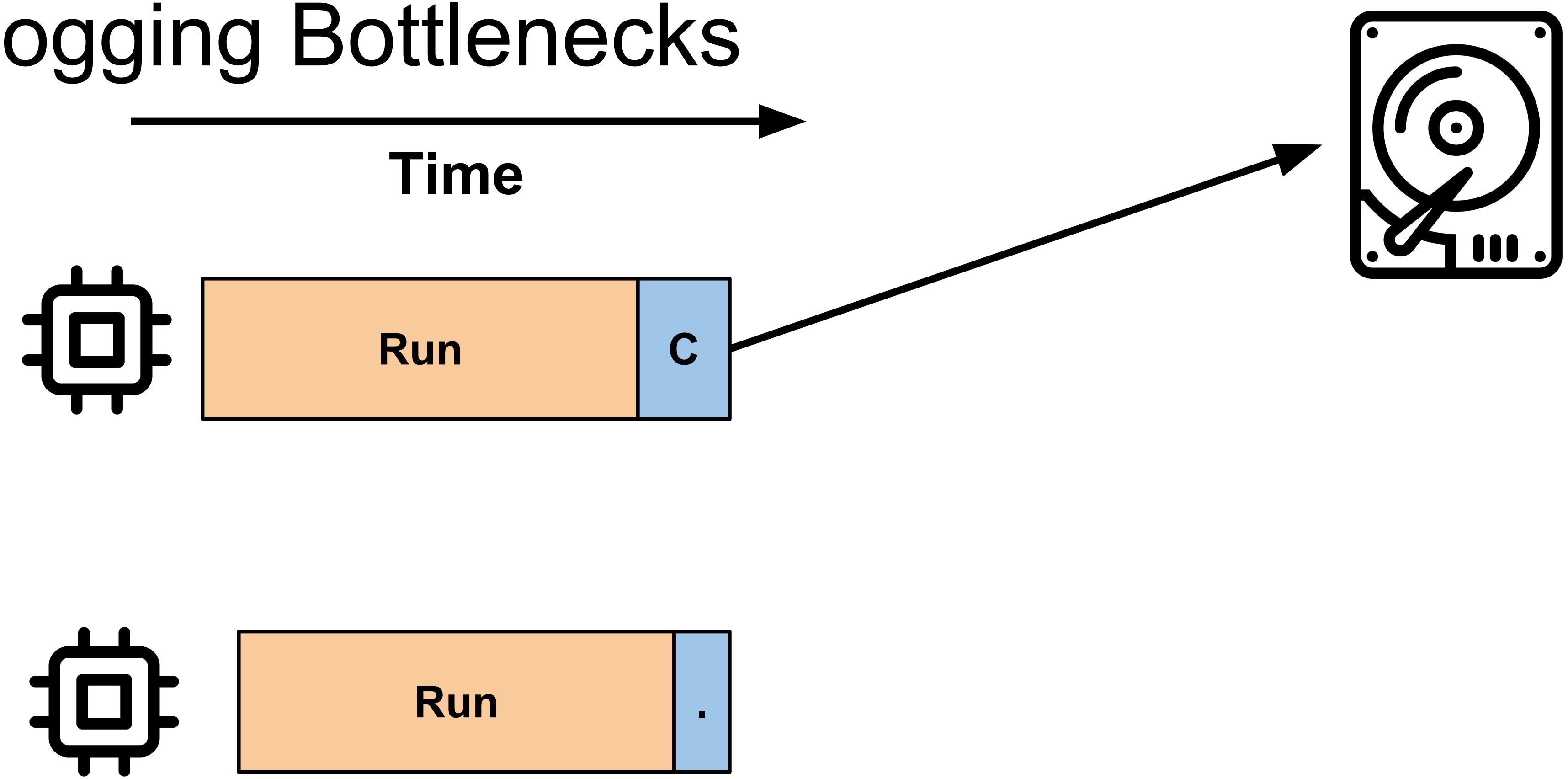


UNIVERSITY OF WATERLOO

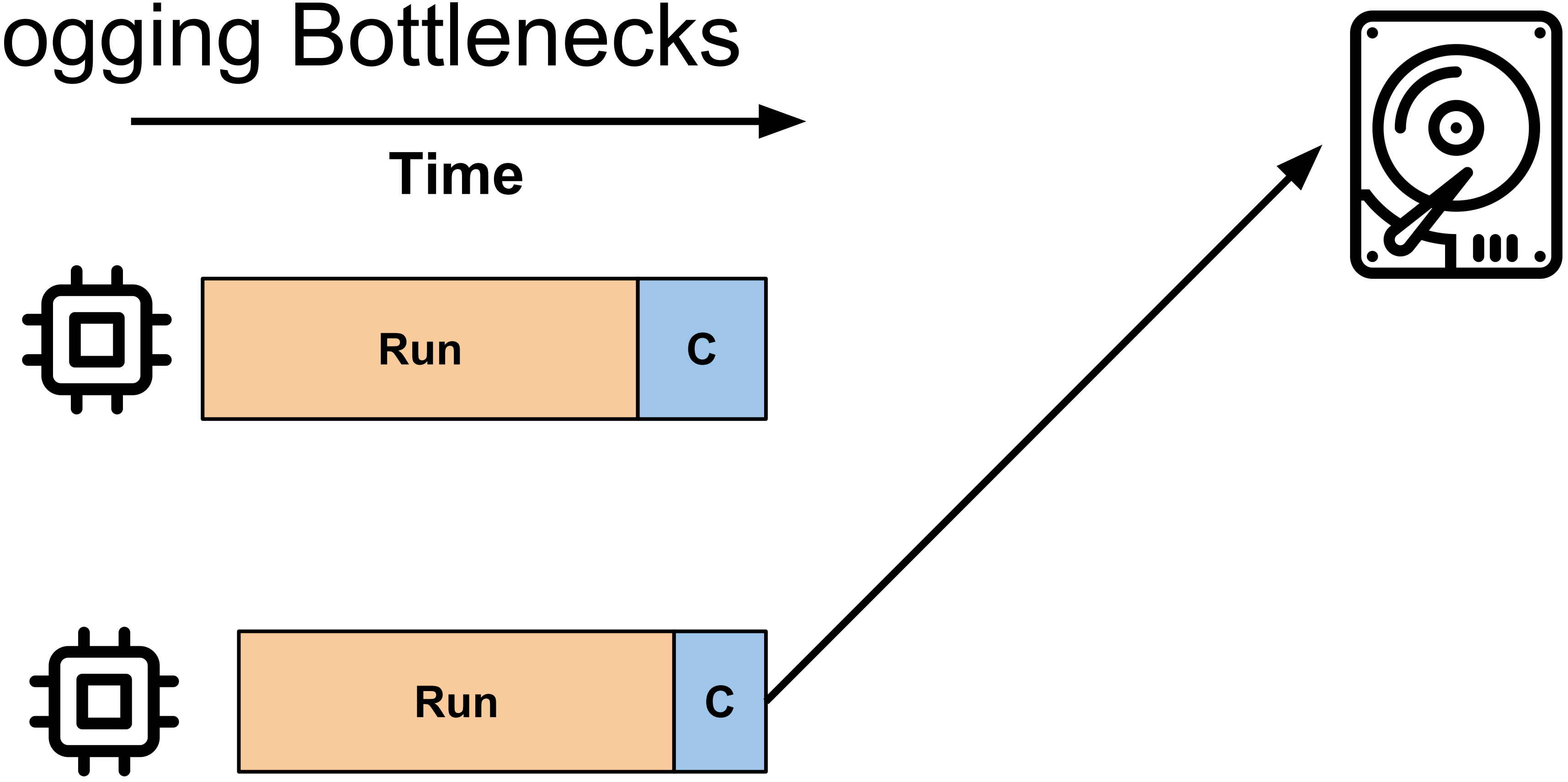
Logging Bottlenecks



Logging Bottlenecks

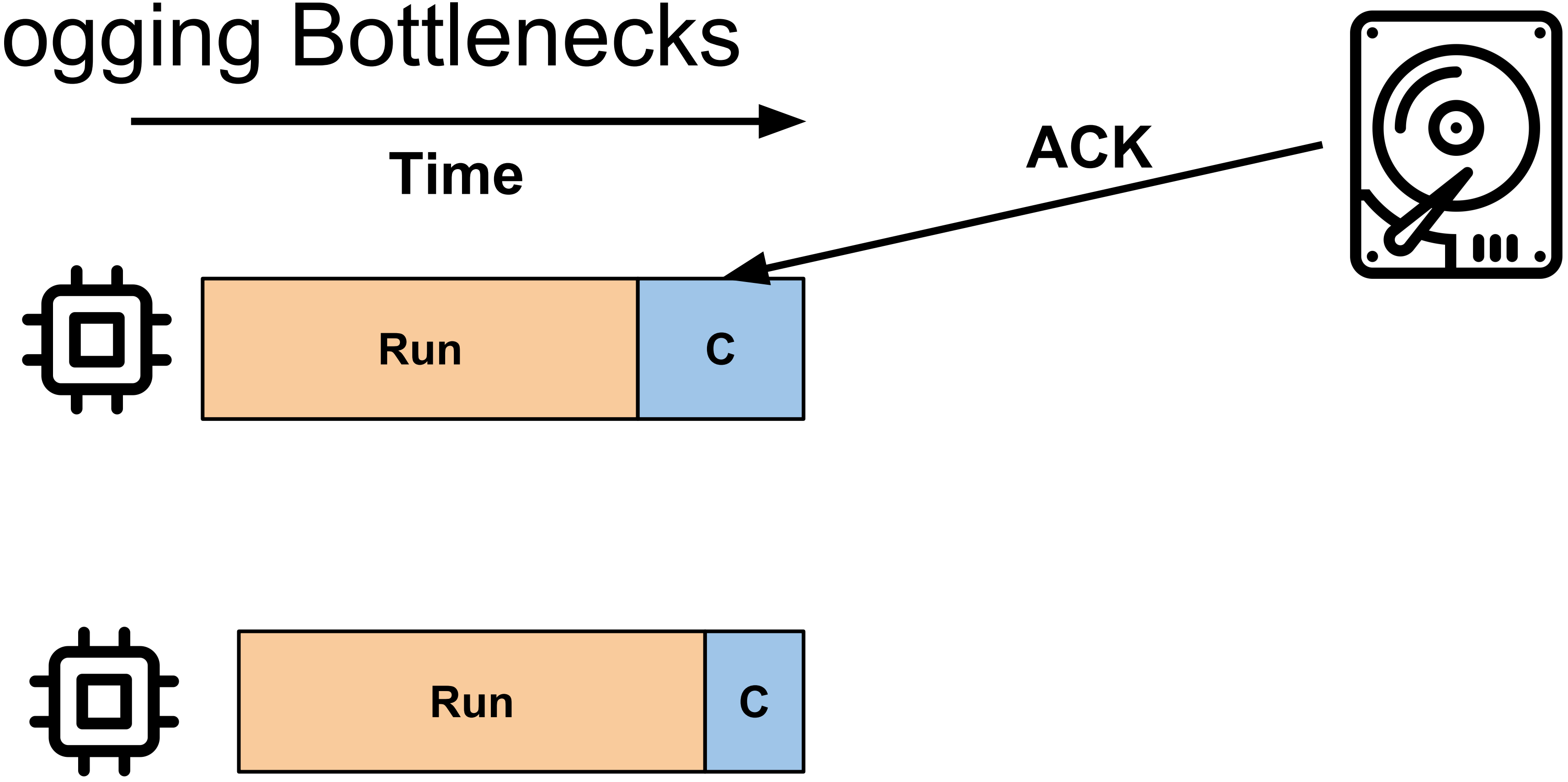


Logging Bottlenecks

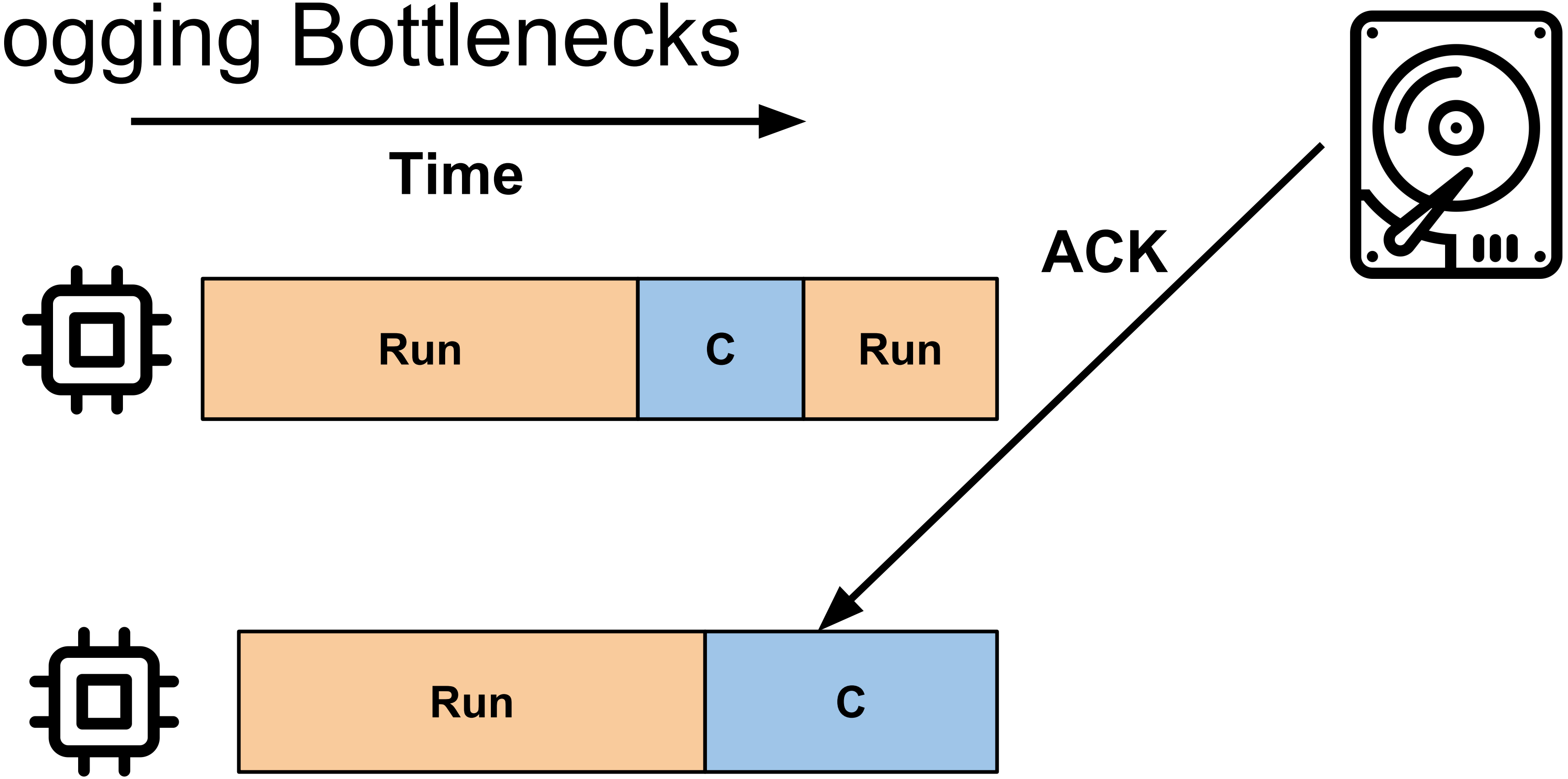


UNIVERSITY OF WATERLOO

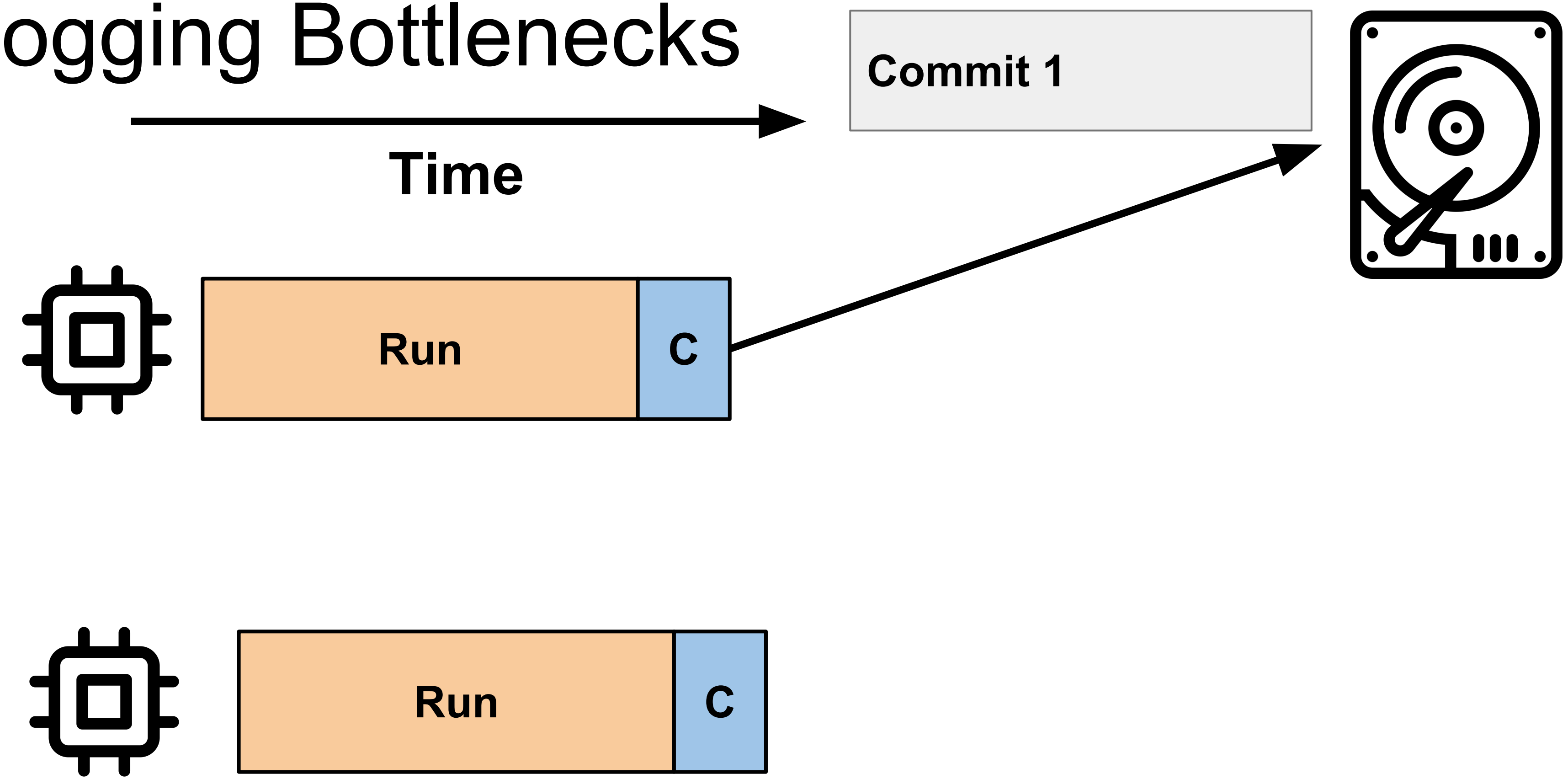
Logging Bottlenecks



Logging Bottlenecks

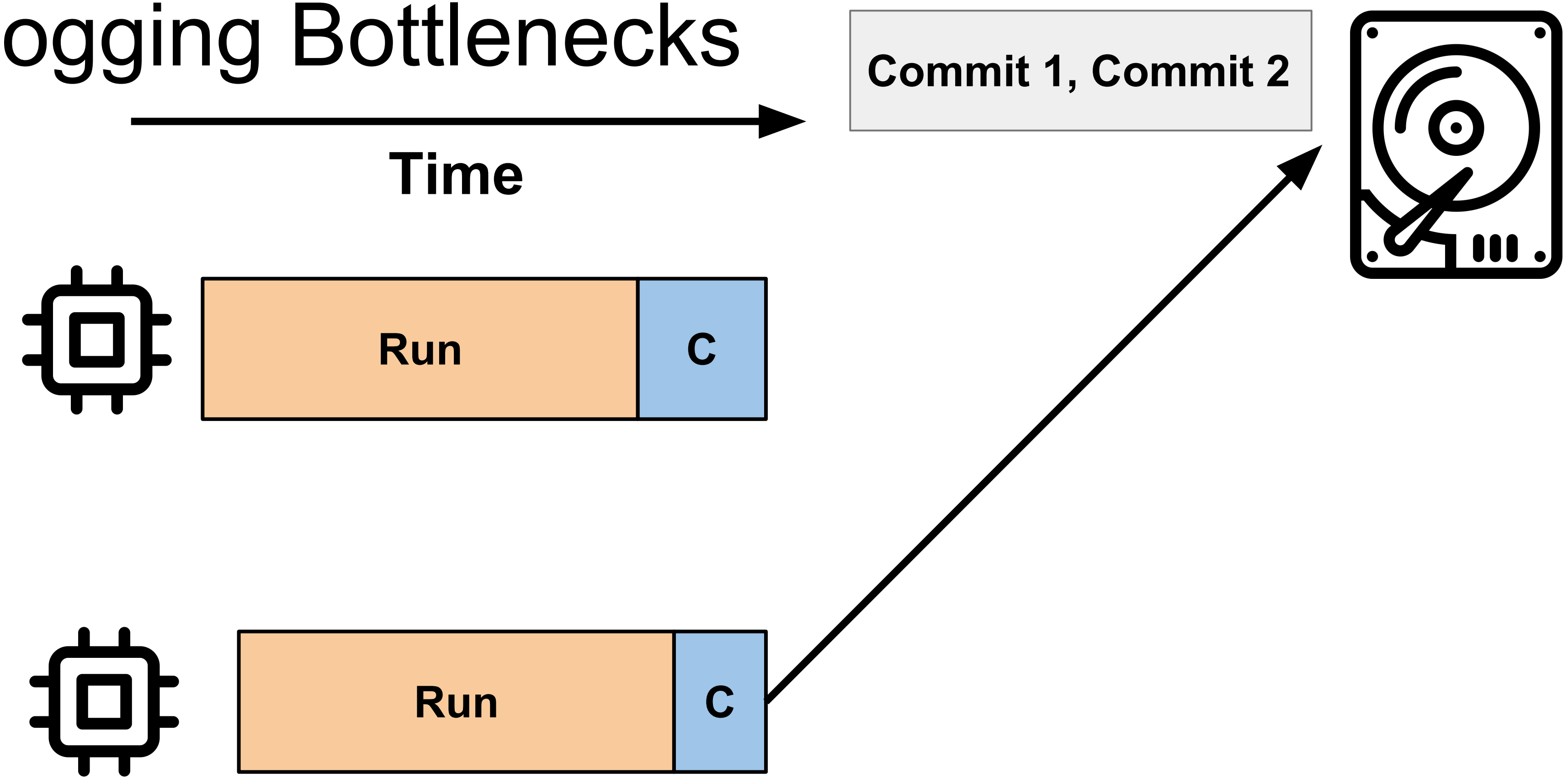


Logging Bottlenecks

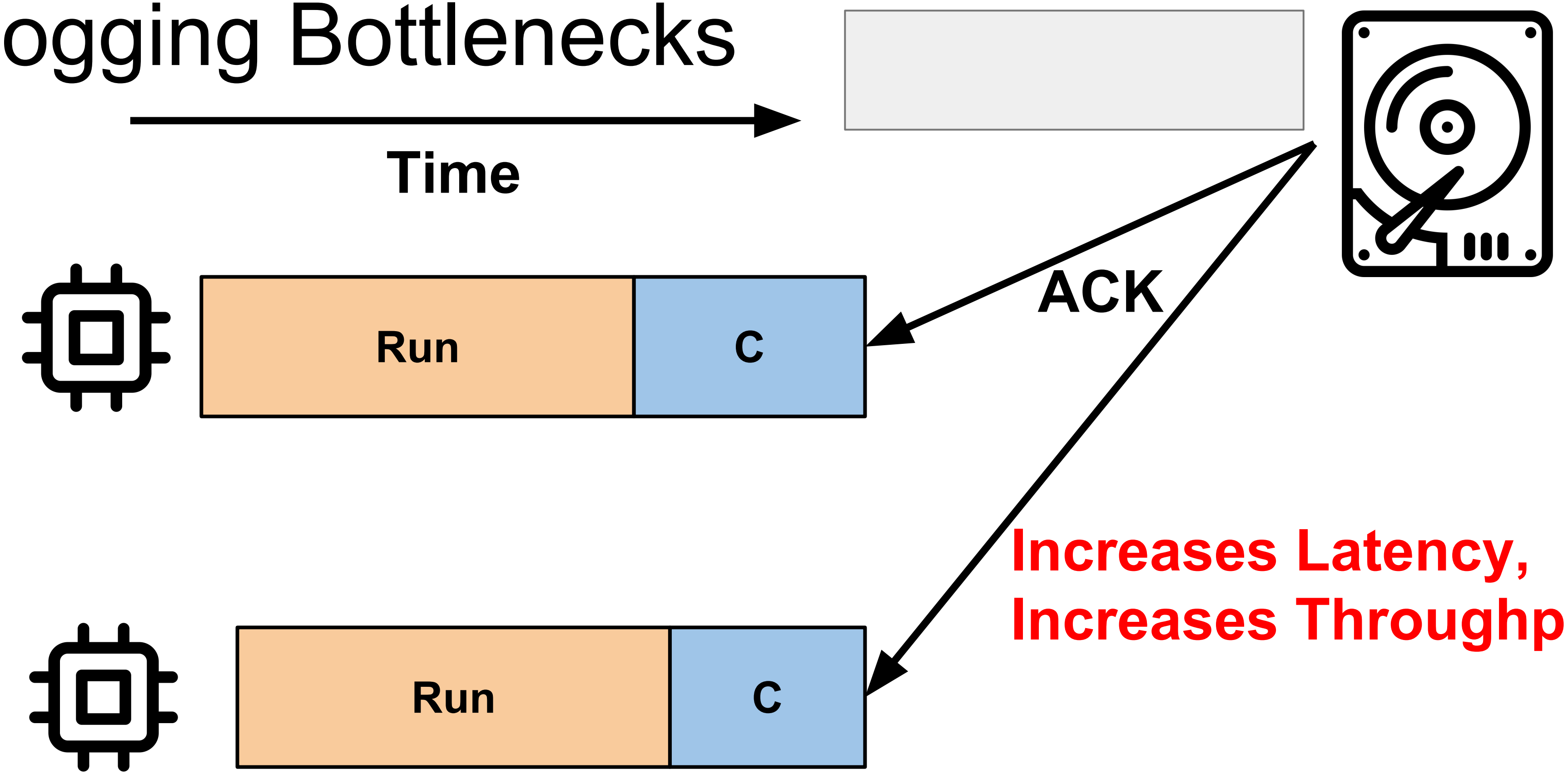


UNIVERSITY OF WATERLOO

Logging Bottlenecks



Logging Bottlenecks



Increases Latency, Increases Throughput

Evaluation

OLTP

TPC-C Order Entry Benchmark

+

OLAP

TPC-H Analysis Benchmark

OLTP Throughput

OLTP

HyPer (single thread)	VoltDB (single node)
126 K	55 K

Why?

OLTP/OLAP Throughput

OLTP
5x

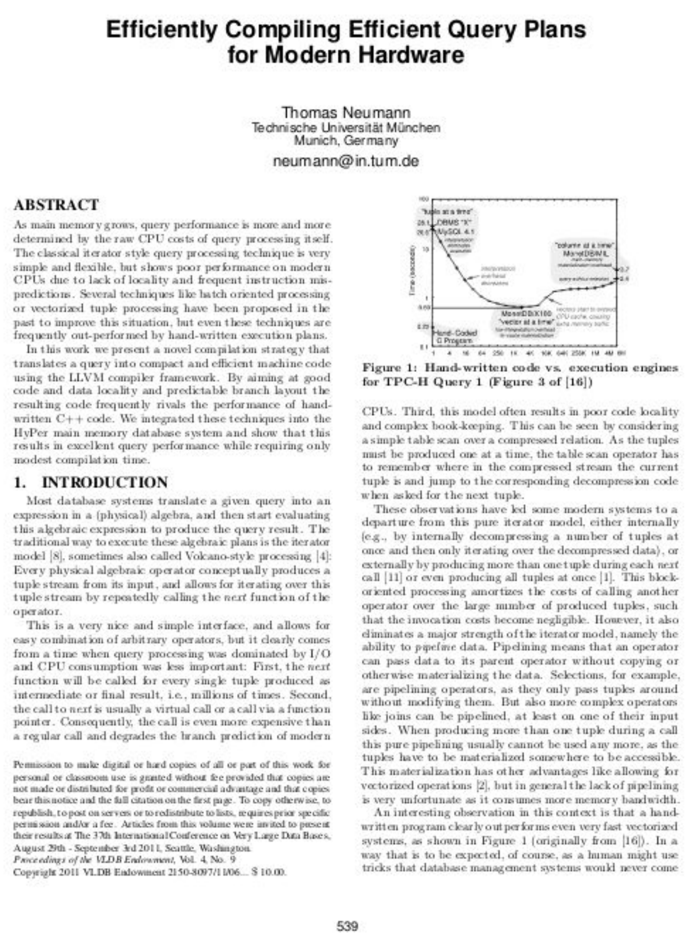
OLAP
3x

HyPer	**VoltDB**
380 K	**300 K**

Competitive with MonetDB

Key Contribution

Leverage OS fork() to make efficient snapshots!



Efficiently Compiling Efficient Query Plans for Modern Hardware

Thomas Neumann
Technische Universität München
Munich, Germany
neumann@in.tum.de

ABSTRACT

As main memory grows, query performance is more and more determined by the raw CPU costs of query processing itself. The classical iterator style query processing technique is very simple and flexible, but shows poor performance on modern CPUs due to lack of locality and frequent instruction mispredictions. Several techniques like batch oriented processing or vectorized tuple processing have been proposed in the past to improve this situation, but even these techniques are frequently out-performed by hand-written execution plans.

In this work we present a novel compilation strategy that translates a query into compact and efficient machine code using the LLVM compiler framework. By aiming at good code and data locality and predictable branch layout the resulting code frequently rivals the performance of hand-written C++ code. We integrated these techniques into the HyPer main memory database system and show that this results in excellent query performance while requiring only modest compilation time.

1. INTRODUCTION

Most database systems translate a given query into an expression in a (physical) algebra, and then start evaluating this algebraic expression to produce the query result. The traditional way to execute these algebraic plans is the iterator model [8], sometimes also called Volcano-style processing [4]: Every physical algebraic operator conceptually produces a tuple stream from its input, and allows for iterating over this tuple stream by repeatedly calling the *next* function of the operator.

This is a very nice and simple interface, and allows for easy combination of arbitrary operators, but it clearly comes from a time when query processing was dominated by I/O and CPU consumption was less important: First, the *next* function will be called for every single tuple produced as intermediate or final result, i.e., millions of times. Second, the call to *next* is usually a virtual call or a call via a function pointer. Consequently, the call is even more expensive than a regular call and degrades the branch prediction of modern

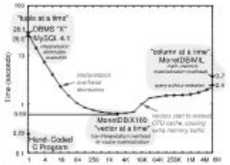


Figure 1: Hand-written code vs. execution engines for TPC-H Query 1 (Figure 3 of [16])

CPUs. Third, this model often results in poor code locality and complex book-keeping. This can be seen by considering a simple table scan over a compressed relation. As the tuples must be produced one at a time, the table scan operator has to remember where in the compressed stream the current tuple is and jump to the corresponding decompression code when asked for the next tuple.

These observations have led some modern systems to a departure from this pure iterator model, either internally (e.g., by internally decompressing a number of tuples at once and then only iterating over the decompressed data), or externally by producing more than one tuple during each *next* call [11] or even producing all tuples at once [1]. This block-oriented processing amortizes the costs of calling another operator over the large number of produced tuples, such that the invocation costs become negligible. However, it also eliminates a major strength of the iterator model, namely the ability to *pipeline* data. Pipelining means that an operator can pass data to its parent operator without copying or otherwise materializing the data. Selections, for example, are pipelining operators, as they only pass tuples around without modifying them. But also more complex operators like joins can be pipelined, at least on one of their input sides. When producing more than one tuple during a call this pure pipelining usually cannot be used any more, as the tuples have to be materialized somewhere to be accessible. This materialization has other advantages like allowing for vectorized operations [2], but in general the lack of pipelining is very unfortunate as it consumes more memory bandwidth.

An interesting observation in this context is that a hand-written program clearly outperforms even very fast vectorized systems, as shown in Figure 1 (originally from [16]). In a way that is to be expected, of course, as a human might use tricks that database management systems would never come

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

Thomas Neumann Tobias Mühlbauer Alfons Kemper
Technische Universität München
{neumann, muehlbau, kemper}@in.tum.de

ABSTRACT

Multi-Version Concurrency Control (MVCC) is a widely employed concurrency control mechanism, as it allows for execution modes where readers never block writers. However, most systems implement only snapshot isolation (SI) instead of full serializability. Adding serializability guarantees to existing SI implementations tends to be *prohibitively expensive*.

We present a novel MVCC implementation for main-memory database systems that has very little overhead compared to serial execution with single-version concurrency control, even when maintaining serializability guarantees. Updating data in-place and storing versions as before-image deltas in undo buffers not only allows us to retain the high scan performance of single-version systems but also forms the basis of our cheap and fine-grained serializability validation mechanism. The novel idea is based on an adaptation of precision locking and verifies that the (extensional) writes of recently committed transactions do not intersect with the (intensional) read predicate space of a committing transaction. We experimentally show that our MVCC model allows very fast processing of transactions with point accesses *as well as* read-heavy transactions and that there is little need to prefer SI over full serializability any longer.

Categories and Subject Descriptors

H.2 [**Database Management**]: Systems

Keywords

Multi-Version Concurrency Control; MVCC; Serializability

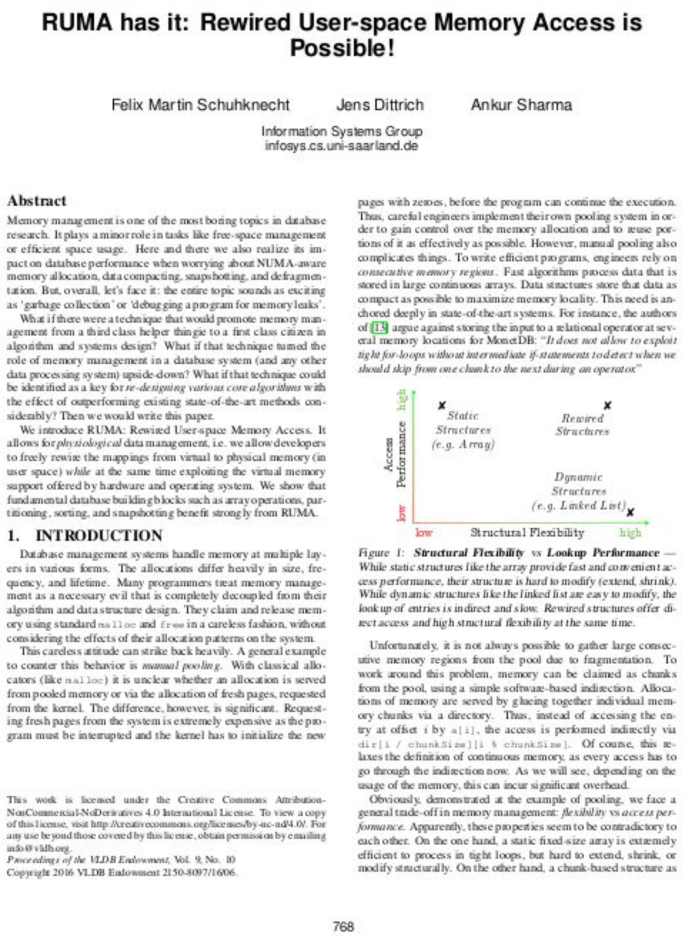
1. INTRODUCTION

Transaction isolation is one of the most fundamental features offered by a database management system (DBMS). It provides the user with the illusion of being alone in the database system, even in the presence of multiple concurrent users, which greatly simplifies application development. In the background, the DBMS ensures that the resulting concurrent access patterns are safe, ideally by being serializable.

Serializability is a great concept, but it is hard to implement efficiently. A classical way to ensure serializability is to rely on a variant of *Two-Phase Locking* (2PL) [42]. Using 2PL, the DBMS maintains read and write locks to ensure that conflicting transactions are executed in a well-defined order, which results in serializable execution schedules. Locking, however, has several major disadvantages: First, readers and writers block each other. Second, most transactions are read-only [33] and therefore harmless from a transaction-ordering perspective. Using a locking-based isolation mechanism, no update transaction is allowed to change a data object that has been read by a potentially long-running read transaction and thus has to wait until the read transaction finishes. This severely limits the degree of concurrency in the system.

Multi-Version Concurrency Control (MVCC) [42, 3, 28] offers an elegant solution to this problem. Instead of updating data objects in-place, each update creates a new version of that data object, such that concurrent readers can still see the old version while the update transaction proceeds concurrently. As a consequence, read-only transactions never have to wait, and in fact do not have to use locking at all. This is an extremely desirable property and the reason why many DBMSs implement MVCC, e.g., Oracle, Microsoft SQL Server [8, 23], SAP HANA [10, 37], and PostgreSQL [34]. However, most systems that use MVCC do not guarantee serializability, but the weaker isolation level *Snapshot Isolation* (SI). Under SI, every transaction sees the database in a certain state (typically the last committed state at the beginning of the transaction) and the DBMS ensures that two concurrent transactions do not update the same data object. Although SI offers fairly good isolation, some non-serializable schedules are still allowed [1, 2]. This is often reluctantly accepted because making SI serializable tends to be *prohibitively expensive* [7]. In particular, the known solutions require keeping track of the entire read set of every transaction, which creates a huge overhead for read-heavy (e.g., analytical) workloads. Still, it is desirable to detect serializability conflicts as they can lead to silent data corruption, which in turn can cause hard-to-detect bugs.

In this paper we introduce a novel way to implement MVCC that is very fast and efficient, both for SI and for full serializability. Our SI implementation is admittedly more carefully engineered than totally new, as MVCC is a well understood approach that recently received renewed interest in the context of main-memory DBMSs [23]. Careful engineering, however, matters as the performance of version maintenance greatly affects transaction *and* query processing. It



RUMA has it: Rewired User-space Memory Access is Possible!

Felix Martin Schuhknecht Jens Dittrich Ankur Sharma

Information Systems Group
infosys.cs.uni-saarland.de

Abstract

Memory management is one of the most boring topics in database research. It plays a minor role in tasks like free-space management or efficient space usage. Here and there we also realize its impact on database performance when worrying about NUMA-aware memory allocation, data compacting, snapshotting, and defragmentation. But, overall, let's face it: the entire topic sounds as exciting as 'garbage collection' or 'debugging a program for memory leaks'.

What if there were a technique that would promote memory management from a third class helper thingie to a first class citizen in algorithm and systems design? What if that technique turned the role of memory management in a database system (and any other data processing system) upside-down? What if that technique could be identified as a key for *re-designing various core algorithms* with the effect of outperforming existing state-of-the-art methods considerably? Then we would write this paper.

We introduce RUMA: Rewired User-space Memory Access. It allows for *physiological* data management, i.e. we allow developers to freely rewire the mappings from virtual to physical memory (in user space) *while* at the same time exploiting the virtual memory support offered by hardware and operating system. We show that fundamental database building blocks such as array operations, partitioning, sorting, and snapshotting benefit strongly from RUMA.

1. INTRODUCTION

Database management systems handle memory at multiple layers in various forms. The allocations differ heavily in size, frequency, and lifetime. Many programmers treat memory management as a necessary evil that is completely decoupled from their algorithm and data structure design. They claim and release memory using standard `malloc` and `free` in a careless fashion, without considering the effects of their allocation patterns on the system.

This careless attitude can strike back heavily. A general example to counter this behavior is *manual pooling*. With classical allocators (like `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, however, is significant. Requesting fresh pages from the system is extremely expensive as the program must be interrupted and the kernel has to initialize the new pages with zeroes, before the program can continue the execution. Thus, careful engineers implement their own pooling system in order to gain control over the memory allocation and to reuse portions of it as effectively as possible. However, manual pooling also complicates things. To write efficient programs, engineers rely on *consecutive memory regions*. Fast algorithms process data that is stored in large continuous arrays. Data structures store that data as compact as possible to maximize memory locality. This need is anchored deeply in state-of-the-art systems. For instance, the authors of [13] argue against storing the input to a relational operator at several memory locations for MonetDB: *"It does not allow to exploit tight for-loops without intermediate if-statements to detect when we should skip from one chunk to the next during an operator."*

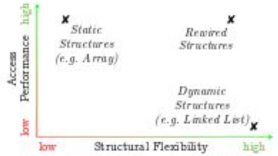


Figure 1: ***Structural Flexibility*** *vs* ***Lookup Performance*** *— While static structures like the array provide fast and convenient access performance, their structure is hard to modify (extend, shrink). While dynamic structures like the linked list are easy to modify, the lookup of entries is indirect and slow. Rewired structures offer direct access and high structural flexibility at the same time.*

Unfortunately, it is not always possible to gather large consecutive memory regions from the pool due to fragmentation. To work around this problem, memory can be claimed as chunks from the pool, using a simple software-based indirection. Allocations of memory are served by glueing together individual memory chunks via a directory. Thus, instead of accessing the entry at offset `i` by `a[i]`, the access is performed indirectly via `dir[i / chunkSize][i % chunkSize]`. Of course, this relaxes the definition of continuous memory, as every access has to go through the indirection now. As we will see, depending on the usage of the memory, this can incur significant overhead.

Obviously, demonstrated at the example of pooling, we face a general trade-off in memory management: *flexibility* vs *access performance*. Apparently, these properties seem to be contradictory to each other. On the one hand, a static fixed-size array is extremely efficient to process in tight loops, but hard to extend, shrink, or modify structurally. On the other hand, a chunk-based structure as

Proceedings of the VLDB Endowment, Vol. 9, No. 10
Copyright 2016 VLDB Endowment 2150-8097/16/06.

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