Isolation as a service
Towards a new decoupled architecture for transaction management

Kevin P. Gaffney, Robert Claus, Jignesh M. Patel
Wisconsin Database Affiliates
The transactional storage manager

Interdependence between components leads to large and complex code.

One of the truly monolithic pieces of a conventional database system.

---

The transactional storage manager
Disadvantages of the monolithic design

The substantial development effort discourages developers of new systems from adding support for transactions.

RocksDB
Transactions were unsupported in initial release, but have since been added\(^2\).

MongoDB
Global database lock was used to serialize operations prior to 2.2 release\(^3\).

The transactional storage manager
Disadvantages of the monolithic design

Complexity may lead to bugs and unexpected behavior.

**PostgreSQL 12.3**: G2-item anomaly (violation of serializability) in serializable mode discovered by *Elle*, a transaction isolation checker⁴.

**MySQL**: Switching from read uncommitted to serializable isolation increases performance under some conditions.

---

Modularizing transaction isolation

Atomicity
Log manager, concurrency control

Consistency
Access methods, runtime checks

Durability
Log manager, recovery manager

Transactional storage manager

DIBS
Database Isolation By Scheduling
Outline

1. Optimizing predicate locking for high throughput
2. Modular transaction scheduling
3. Evaluation and use cases
4. Discussion
Predicate locking

Predicate locking was originally proposed as a solution to the **phantom read** problem.

We focus on an additional benefit of predicate locking: **independence from the contents and implementation of the database system**.
Predicate locking

Definition

A predicate lock is defined a tuple $L = (R, P, a)$ where

- $R$ is a relation,
- $P$ is a predicate that can be evaluated on a tuple $t \in R$,
- $a$ is the access mode (*read* or *write*).
Predicate locking

Conflict

Two predicate locks $L = (R, P, a)$ and $L' = (R', P', a')$ are said to conflict if all of the following are true:

1. $R = R'$
2. $a = \text{write}$ or $a' = \text{write}$
3. There exists some feasible tuple $t \in R$ such that $P(t) \land P'(t) = \text{TRUE}$. 
Predicate locking
Sources of overhead

Checking $P(t) \land P'(t) = TRUE$ can be reduced to the boolean satisfiability problem (SAT), which is NP-complete.

A new predicate lock must be compared to every other active predicate lock to determine whether it conflicts. This requires synchronization between threads.

The following optimizations reduce the overhead of predicate locking.
Optimization I
Conjunct grouping

Goal: reduce exponential blowup of converting to DNF.

Observation: Each term of a DNF predicate must contain at least two references to the same column to be a contradiction.

For example,

\[(a = v_1 \land a = v_2)\] is a contradiction if \(v_1 \neq v_2\).

\[(a = v_1 \land b = v_2)\] is never a contradiction (satisfiable for all \(v_1\) and \(v_2\)).
Optimization I
Conjunct grouping

First, separate the conjuncts of $P$ and $P'$ into groups that access disjoint sets of columns.

Then, convert each group to DNF individually and check for conflict.

For example,

$$P \equiv ((a = 1 \lor a = 2) \land (b = 1 \lor b = 2))$$

$$P' \equiv ((a = 3 \lor a = 4) \land (b = 3 \lor b = 4)).$$
Goal: reduce latency of checking for conflict.

Applications commonly use prepared statements to reduce overhead of parsing and optimization and protect against SQL injection.

Similarly, parameterized predicates can be analyzed offline, reducing online overhead.
Optimization II
Prepared predicates

Define a conflict graph that includes one vertex for each parameterized predicate.

At each edge, store a conflict predicate that evaluates to TRUE if the predicates conflict.

For example,

\[ P \equiv (a = v_1 \lor b > v_2) \]

\[ P' \equiv (a = v_3 \land b = v_4) \]
Optimization III
Column filtering

Goal: reduce contention for centralized set of predicate locks.

Access to the centralized set of predicate locks must be synchronized among threads.

Acquiring and releasing a predicate lock is a severe scalability bottleneck.

\[
L_1 = (R_1, P_1, a_1) \\
L_2 = (R_2, P_2, a_2) \\
L_3 = (R_3, P_3, a_3) \\
\vdots
\]
Instead of a single set, construct $N$ buckets of active predicate locks.

Define function $f$ that maps each predicate’s values to a subset of the buckets, ensuring that predicates that could potentially conflict are mapped to a common bucket.

For example,

$$P \equiv (a = 1 \land b = 1)$$
$$P' \equiv (a = 2 \land b = 1)$$

Please see paper for discussion on choice of $N$ and $f$. 
Database Isolation By Scheduling (DIBS)

Architecture

Client Applications

DIBS

Predicate Lock Manager (PLM)

Database Connector

begin() : void
commit() : void
rollback() : void
savepoint(id: string) : void
rollbackTo(id : string) void
execute(request: Request) : void

Worker Threads

Client Connectors

getNextTransaction() : Transaction
commitTransaction(t: Transaction) : void

Data Platform
Workloads

**TATP**

Relational online transaction processing (OLTP) benchmark.

Simulates a telecommunications application.

1 million “subscriber” records.

**YCSB**

Key-value OLTP benchmark consisting of point reads and updates.

Read intensive = 95/5.

Write intensive = 50/50.

1 million records.

**SubscriberScan**

Benchmark with scans using complex, CNF predicates.

Designed to test exponential blowup of converting to DNF.

Varying number of conjuncts from 1 to 8.
Use case I
Isolation as a service for new systems

We evaluate DIBS as the sole isolation mechanism for a prototype system built on Apache Arrow\(^5\).

Arrow is a cross-platform columnar data format.

Benchmark transactions were hard-coded and compiled to LLVM.

---

Use case I
Isolation as a service for new systems

We evaluate the performance impact of cumulatively adding each predicate locking optimization.
TATP on Arrow

Naive predicate locking does not scale.
Latch contention is the bottleneck, so O1 and O2 do not improve scaling.
TATP on Arrow

Latch contention is the bottleneck, so O1 and O2 do not improve scaling.
TATP on Arrow

O3 reduces contention and improves scaling.
Naive predicate locking suffers from exponential-time conversion to DNF.
SubscriberScan on Arrow

O1 reduces overhead of converting to DNF.

The chart shows the throughput (K tps) against the number of conjuncts. The legend indicates two lines:
- **Naive**
- **O1: Conjunct grouping**
SubscriberScan on Arrow

O2 provides an additional modest performance increase.
SubscriberScan on Arrow

O3 has no effect for this workload.
Use case II
Transaction merging

The DIBS API includes a commit hook that a data platform can use to flush transaction logs and clean up resources.

**Standard mode**
Commit hook called at the end of each client transaction.

**Transaction merging mode**
Commit hook delayed until multiple client transactions have been completed.
TATP on SQLite with transaction merging

 SQLite’s performance is bottlenecked by log flushing and does not scale.
TATP on SQLite with transaction merging

Transaction merging with DIBS nearly triples throughput.
Use case III
Column-granularity locking

Two predicate locks $L = (R, P, a)$ and $L' = (R', P', a')$ are said to conflict if all of the following are true:

1. $R = R'$
2. $a = \text{write}$ or $a' = \text{write}$
3. There exists some feasible tuple $t \in R$ such that $P(t) \land P'(t) = \text{TRUE}$.

Change to: Intersecting columns are not read-only.
Experiments on MySQL with column-granularity locking

**Integrated serializable:**
MySQL serializable isolation without DIBS.

**DIBS serializable:**
MySQL read uncommitted isolation with DIBS (results in serializable isolation).
YCSB on MySQL with column-granularity locking

MySQL’s serializable isolation is bottlenecked by row contention.

Throughput (K tps)

Worker threads

Skew = 2.0

Integrated serializable
YCSB on MySQL with column-granularity locking

DIBS reduces row contention, increasing performance by 40%.

Skew = 2.0
YCSB on MySQL with column-granularity locking

Performance suffers from contention for rows.

Worker threads = 20

Integrated serializable  DIBS serializable
YCSB on MySQL with column-granularity locking

At high skew, DIBS increases performance by 30-40%.

Worker threads = 20
More info in the full paper

Supporting experiments
SEATS benchmark on Arrow, YCSB on MySQL with transaction merging, transaction latency, memory-optimized SQL Server baseline.

Further discussion
Tradeoffs surrounding column filtering optimization, extensions to more complex predicates, assumptions about the data platform, changes needed to support additional isolation levels.

Acknowledgements

This work was supported in part by Semiconductor Research Corporation (SRC) and the National Science Foundation (NSF).
Thank you!

Please feel free to reach out to me on gather.town or via email: kpgaffney@wisc.edu

Kevin P. Gaffney  Robert Claus  Jignesh M. Patel
Non-PK access microbenchmark

0% non-PK

1% non-PK

5% non-PK

\[ N \text{ (number of buckets)} \]

- 1
- 2
- 4
- 8
- 16
- 32
- 64
- 128
Memory-optimized SQL Server benchmark performance

Throughput (K tps)

TATP

SEATS

SubscriberScan, 6 conjuncts per predicate
Combining optimizations

Insert $P$ into buckets.
Can $P$ be mapped with $f(P)$?

- **Y**: Insert into buckets $f(P)$
- **N**: Insert into all $N$ buckets

Evaluate against other $P'$ in bucket(s).
Are both $P$ and $P'$ prepared predicates?

- **Y**: Evaluate prepared predicate
- **N**: Evaluate with conjunct grouping

**40**