Multi-Version Concurrency Control (Garbage Collection)

@Andy_Pavlo // 15-721 // Spring 2020
A MVCC DBMS needs to remove **reclaimable** physical versions from the database over time.

→ No active txn in the DBMS can “see” that version (SI).

→ The version was created by an aborted txn.

The DBMS uses the tuples' version meta-data to decide whether it is visible.
OBSERVATION

We have assumed that queries / txns will complete in a short amount of time. This means that the lifetime of an obsolete version is short as well.

But HTAP workloads may have long running queries that access old snapshots. Such queries block the traditional garbage collection methods that we have discussed.
PROBLEMS WITH OLD VERSIONS

- Increased Memory Usage
- Memory Allocator Contention
- Longer Version Chains
- Garbage Collector CPU Spikes
- Poor Time-based Version Locality
TODAY’S AGENDA

MVCC Deletes
Garbage Collection
Block Compaction
The DBMS physically deletes a tuple from the database only when all versions of a logically deleted tuple are not visible.
→ If a tuple is deleted, then there cannot be a new version of that tuple after the newest version.
→ No write-write conflicts / first-writer wins

We need a way to denote that tuple has been logically delete at some point in time.
MVCC DELETES

Approach #1: Deleted Flag
→ Maintain a flag to indicate that the logical tuple has been deleted after the newest physical version.
→ Can either be in tuple header or a separate column.

Approach #2: Tombstone Tuple
→ Create an empty physical version to indicate that a logical tuple is deleted.
→ Use a separate pool for tombstone tuples with only a special bit pattern in version chain pointer to reduce the storage overhead.
GC DESIGN DECISIONS

Index Clean-up
Version Tracking Level
Frequency
Granularity
Comparison Unit
GC – INDEX CLEAN-UP

The DBMS must remove a tuples' keys from indexes when their corresponding versions are no longer visible to active txns.

Track the txn's modifications to individual indexes to support GC of older versions on commit and removal modifications on abort.
PELOTON MISTAKE

Thread #1

**Begin @ 10**

**UPDATE(A)**

`key=222`

---

**Index**

- **VERSION**
- **BEGIN-TS**
- **END-TS**
- **KEY**

<table>
<thead>
<tr>
<th>VERSION</th>
<th>BEGIN-TS</th>
<th>END-TS</th>
<th>KEY</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₁</td>
<td>1</td>
<td>∞</td>
<td>111</td>
</tr>
</tbody>
</table>
Thread #1

Begin @ 10

UPDATE(A)

key=222

PELOTON MISTAKE

Index

<table>
<thead>
<tr>
<th>VERSION</th>
<th>BEGIN-TS</th>
<th>END-TS</th>
<th>KEY</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_1$</td>
<td>1</td>
<td>$10$</td>
<td>111</td>
</tr>
<tr>
<td>$A_2$</td>
<td>$10$</td>
<td>$\infty$</td>
<td>222</td>
</tr>
</tbody>
</table>
Thread #1

*Begin @ 10*

**PELOTON MISTAKE**

**Index**

**Thread #1**

*Begin @ 10*

**UPDATE(A)**

`key=222`

**UPDATE(A)**

`key=333`

**Version**

<table>
<thead>
<tr>
<th>VERSION</th>
<th>BEGIN-TS</th>
<th>END-TS</th>
<th>KEY</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_1$</td>
<td>1</td>
<td>$1\theta$</td>
<td>111</td>
</tr>
<tr>
<td>$A_2$</td>
<td>$1\theta$</td>
<td>$\infty$</td>
<td>222</td>
</tr>
</tbody>
</table>
If a txn writes to same tuple more than once, then it just overwrites its previous version.
Thread #1

*Begin @ 10*

**PELOTON MISTAKE**

If a txn writes to same tuple more than once, then it just overwrites its previous version.
If a txn writes to same tuple more than once, then it just overwrites its previous version. Upon rollback, the DBMS did not know what keys it added to the index in previous versions.
GC – VERSION TRACKING

**Approach #1: Tuple-level**
→ Find old versions by examining tuples directly.
→ Background Vacuuming vs. Cooperative Cleaning

**Approach #2: Transaction-level**
→ Txns keep track of their old versions so the DBMS does not have to scan tuples to determine visibility.

**Approach #3: Epochs**
→ Group multiple txns togethers into an epoch and then
**GC – VERSION TRACKING**

**Thread #1**

*Begin @ 10*

<table>
<thead>
<tr>
<th></th>
<th>BEGIN-TS</th>
<th>END-TS</th>
<th>DATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₂</td>
<td>1</td>
<td>∞</td>
<td>–</td>
</tr>
<tr>
<td>B₆</td>
<td>8</td>
<td>∞</td>
<td>–</td>
</tr>
</tbody>
</table>
Thread #1
Begin @ 10

UPDATE(A)

GC – VERSION TRACKING

<table>
<thead>
<tr>
<th></th>
<th>BEGIN-TS</th>
<th>END-TS</th>
<th>DATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₂</td>
<td>1</td>
<td>10</td>
<td>-</td>
</tr>
<tr>
<td>B₆</td>
<td>8</td>
<td>∞</td>
<td>-</td>
</tr>
<tr>
<td>A₃</td>
<td>10</td>
<td>∞</td>
<td>-</td>
</tr>
</tbody>
</table>
Thread #1
Begin @ 10

Old Versions
A₂

UPDATE(A)

BEGIN-TS | END-TS | DATA
----------|--------|--------
A₂        | 1      | 10     |
B₆        | 8      | ∞      |
A₃        | 10     | ∞      |
### GC – VERSION TRACKING

#### Thread #1

**Begin @ 10**

<table>
<thead>
<tr>
<th>Old Versions</th>
<th>UPDATE(A)</th>
<th>UPDATE(B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_2$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BEGIN-TS</th>
<th>END-TS</th>
<th>DATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_2$</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>$B_6$</td>
<td>8</td>
<td>$\infty$</td>
</tr>
<tr>
<td>$A_3$</td>
<td>10</td>
<td>$\infty$</td>
</tr>
</tbody>
</table>
### GC – VERSION TRACKING

#### Thread #1

**Begin @ 10**

**Old Versions**

<table>
<thead>
<tr>
<th>Version</th>
<th>BEGIN-TS</th>
<th>END-TS</th>
<th>DATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₂</td>
<td>1</td>
<td>10</td>
<td>-</td>
</tr>
<tr>
<td>B₆</td>
<td>8</td>
<td>10</td>
<td>-</td>
</tr>
<tr>
<td>A₃</td>
<td>10</td>
<td>∞</td>
<td>-</td>
</tr>
<tr>
<td>B₇</td>
<td>10</td>
<td>∞</td>
<td>-</td>
</tr>
</tbody>
</table>
## GC – VERSION TRACKING

**Thread #1**

**Begin @ 10**

<table>
<thead>
<tr>
<th>Old Versions</th>
<th>UPDATE(A)</th>
<th>UPDATE(B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₂</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B₆</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table

<table>
<thead>
<tr>
<th></th>
<th>BEGIN-TS</th>
<th>END-TS</th>
<th>DATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₂</td>
<td>1</td>
<td>10</td>
<td>–</td>
</tr>
<tr>
<td>B₆</td>
<td>8</td>
<td>10</td>
<td>–</td>
</tr>
<tr>
<td>A₃</td>
<td>10</td>
<td>∞</td>
<td>–</td>
</tr>
<tr>
<td>B₇</td>
<td>10</td>
<td>∞</td>
<td>–</td>
</tr>
</tbody>
</table>
**Thread #1**

*Begin @ 10*

*Commit @ 15*

**Old Versions**

- A_2
- B_6

---

**GC – VERSION TRACKING**

<table>
<thead>
<tr>
<th></th>
<th>BEGIN-TS</th>
<th>END-TS</th>
<th>DATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_2</td>
<td>1</td>
<td>15</td>
<td>–</td>
</tr>
<tr>
<td>B_6</td>
<td>8</td>
<td>15</td>
<td>–</td>
</tr>
<tr>
<td>A_3</td>
<td>15</td>
<td>∞</td>
<td>–</td>
</tr>
<tr>
<td>B_7</td>
<td>15</td>
<td>∞</td>
<td>–</td>
</tr>
</tbody>
</table>
GC – VERSION TRACKING

Thread #1

Begin @ 10
Commit @ 15

Old Versions

UPDATE (A)
UPDATE (B)

Vacuum

TS<15

A2
B6

BEGIN-TS | END-TS | DATA
--- | --- | ---
A2 | 1 | 15 | –
B6 | 8 | 15 | –
A3 | 15 | ∞ | –
B7 | 15 | ∞ | –
GC – FREQUENCY

How often the DBMS should invoke the GC procedure to remove versions.

Need to balance many factors:
→ Too frequent will waste cycles and slow down txns.
→ Too infrequent will cause storage overhead to increase and increase the length of version chains.
GC – FREQUENCY

Approach #1: Periodically
→ Run the GC at fixed intervals or when some threshold has been met (e.g., epoch, memory limits).
→ Some DBMSs can adjust this interval based on load.

Approach #2: Continuously
→ Run the GC as part of the regular txn processing (e.g., on commit, during query execution).
GC – GRANULARITY

How should the DBMS internally organize the expired versions that it needs to check to determine whether they are reclaimable.

Trade-off between the ability to reclaim versions sooner versus computational overhead.
Approach #1: Single Version
→ Track the visibility of individual versions and reclaim them separately.
→ More fine-grained control, but higher overhead.

Approach #2: Group Version
→ Organize versions into groups and reclaim all of them together.
→ Less overhead but may delay reclaims.
Approach #3: Tables

→ Reclaim all versions from a table if the DBMS determines that active txns will never access it.

→ Special case for stored procedures and prepared statements since it requires the DBMS knowing what tables a txn will access in advance.
How should the DBMS determine whether version(s) are reclaimable.

Examining the list of active txns and reclaimable versions should be latch-free.
→ It is okay if the GC misses a recently committed txn. It will find it in the next round.
GC – COMPARISON UNIT

Approach #1: Timestamp
→ Use a global minimum timestamp to determine whether versions are safe to reclaim.
→ Easiest to implement and execute.

Approach #2: Interval
→ Excise timestamp ranges that are not visible.
→ More difficult to identify ranges.
## GC – COMPARISON UNIT

### Thread #1
**Begin @ 10**
- **READ(A)**

### Thread #2
**Begin @ 20**
- **Commit @ 25**
- **UPDATE(A)**

### Thread #3
**Begin @ 30**
- **Commit @ 35**
- **UPDATE(A)**

### Table
<table>
<thead>
<tr>
<th></th>
<th>BEGIN-TS</th>
<th>END-TS</th>
<th>DATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₁</td>
<td>1</td>
<td>25</td>
<td>-</td>
</tr>
<tr>
<td>A₂</td>
<td>25</td>
<td>35</td>
<td>-</td>
</tr>
<tr>
<td>A₃</td>
<td>35</td>
<td>∞</td>
<td>-</td>
</tr>
</tbody>
</table>

### Timestamp
- GC cannot reclaim **A₂** because the lowest active txn TS (10) is less than END-TS.

### Interval
- GC can reclaim **A₂** because no active txn TS intersects the interval [25,35].
Main Data Table

<table>
<thead>
<tr>
<th>Thread #1</th>
<th>Begin @ 15</th>
</tr>
</thead>
<tbody>
<tr>
<td>A\text{\textsubscript{50}}</td>
<td>(ATTR1→Andy, ATTR2→$99)</td>
</tr>
<tr>
<td>Thread #2</td>
<td>Begin @ 55</td>
</tr>
<tr>
<td>A\text{\textsubscript{50}}</td>
<td>(ATTR2→$99)</td>
</tr>
</tbody>
</table>

Delta Storage

| A\text{\textsubscript{50}} | (ATTR2→$99) |
| A\text{\textsubscript{40}} | (ATTR2→$88) |
| A\text{\textsubscript{30}} | (ATTR2→$77) |
| A\text{\textsubscript{20}} | (ATTR1→Andy) |
| A\text{\textsubscript{10}} | (ATTR2→$66) Ø |
OBSERVATION

If the application deletes a tuple, then what should the DBMS do with the slots occupied by that tuple's versions?
→ Always reuse variable-length data slots.
→ More nuanced for fixed-length data slots.

What if the application deletes many (but not all) tuples in a table in a short amount of time?
MVCC DELETED TUPLES

Approach #1: Reuse Slot
→ Allow workers to insert new tuples in the empty slots.
→ Obvious choice for append-only storage since there is no distinction between versions.
→ Destroys temporal locality of tuples in delta storage.

Approach #2: Leave Slot Unoccupied
→ Workers can only insert new tuples in slots that were not previously occupied.
→ Ensures that tuples in the same block were inserted into the database at around the same time.
→ Need an extra mechanism to fill holes.
CONSOLIDATING LESS-THAN-FULL BLOCKS INTO FEWER BLOCKS AND THEN RETURNING MEMORY TO THE OS.
→ Move data using **DELETE + INSERT** to ensure transactional guarantees during consolidation.

IDEALLY THE DBMS WILL WANT TO STORE TUPLES THAT ARE LIKELY TO BE ACCESSED TOGETHER WITHIN A WINDOW OF TIME TOGETHER IN THE SAME BLOCK.
→ This will matter more when we talk about compression and moving cold data out to disk.
**BLOCK COMPACCTION – TARGETS**

Approach #1: Time Since Last Update
→ Leverage the BEGIN-TS in each tuple.

Approach #2: Time Since Last Access
→ Expensive to maintain unless tuple has READ-TS.

Approach #3: Application-level Semantics
→ Tuples from the same table that are related to each other according to some higher-level construct.
→ Difficult to figure out automatically.
BLOCK COMPACTION – TRUNCATE

**TRUNCATE** operation removes all tuples in a table.
→ Think of it like a **DELETE** without a **WHERE** clause.

Fastest way to execute is to drop the table and then create it again.
→ Do not need to track the visibility of individual tuples.
→ The GC will free all memory when there are no active txns that exist before the drop operation.
→ If the catalog is transactional, then this easy to do.
PARTING THOUGHTS

Classic storage vs. compute trade-off.

My impression is that people want to reduce the memory footprint of the DBMS and are willing to pay a (small) computational overhead for more aggressive GC.
Consider a program with functions `foo` and `bar`.

How can we speed it up with only a debugger?
→ Randomly pause it during execution
→ Collect the function call stack
RANDOM PAUSE METHOD

Consider this scenario
→ Collected 10 call stack samples
→ Say 6 out of the 10 samples were in foo

What percentage of time was spent in foo?
→ Roughly 60% of the time was spent in foo
→ Accuracy increases with # of samples
Say we optimized *foo* to run two times faster
What’s the expected overall speedup?
→ 60% of time spent in *foo* drops in half
→ 40% of time spent in *bar* unaffected

By Amdahl’s law, overall speedup = \( \frac{1}{\frac{p}{s} + (1-p)} \)
→ \( p \) = percentage of time spent in optimized task
→ \( s \) = speed up for the optimized task
→ Overall speedup = \( \frac{1}{\frac{0.6}{2} + 0.4} = 1.4 \) times faster
PROFILING TOOLS FOR REAL

Choice #1: Valgrind
→ Heavyweight binary instrumentation framework with different tools to measure different events.

Choice #2: Perf
→ Lightweight tool that uses hardware counters to capture events during execution.
CHOICE #1: VALGRIND

Instrumentation framework for building dynamic analysis tools.

→ **memcheck**: a memory error detector
→ **callgrind**: a call-graph generating profiler
→ **massif**: memory usage tracking.
Using callgrind to profile the target benchmark and the overall DBMS in general:

```
$ export TERRIER_BENCHMARK_THREADS=16
$ valgrind --tool=callgrind --trace-children=yes ./relwithdebinfopl overslot_iterator_benchmark
```

Profile data visualization tool:

```
$ kcachegrind callgrind.out.12345
```
Using callgrind to profile the target benchmark and the overall DBMS in general:

Profile data visualization tool:

```
$ kcachegrind callgrind.out.12345
```

```
$ export TERRIER_BENCHMARK_THREADS=16
$ valgrind --tool=callgrind --trace-children=yes ./relwithdebinfo/slot_iterator_benchmark
```
Using callgrind to profile the target benchmark and the overall DBMS in general:

Profile data visualization tool:

```
$ kcachegrind callgrind.out.12345
```

```
$ export TERRIER_BENCHMARK_THREADS=16
$ valgrind --tool=callgrind --trace-children=yes .relwithdebinfo/slot_iterator_benchmark
```

Cumulative Time Distribution

Callgraph View
CHOICE #2: PERF

Tool for using the performance counters subsystem in Linux.

→ `-e` = sample the event cycles at the user level only
→ `-c` = collect a sample every 2000 occurrences of event

```
$ export TERRIER_BENCHMARK_THREADS=16
$ perf record -e cycles:u -c 2000
./relwithdebinfo/slot_iterator_benchmark
```

Uses counters for tracking events

→ On counter overflow, the kernel records a sample
→ Sample contains info about program execution
PERF VISUALIZATION

We can also use `perf` to visualize the generated profile for our application.

```
$ perf report
```

There are also third-party visualization tools:
→ `Hotspot`
We can also use `perf` to visualize the generated profile for our application. There are also third-party visualization tools:

```
$ perf report
```

### Cumulative Event Distribution

```
Cumulative Event Distribution

<table>
<thead>
<tr>
<th>Command</th>
<th>Shared Object</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>concurrent_read</td>
<td>concurrent_readBenchmark</td>
<td>[.].ZN7terrier7storage11StorageUtil17CopyWithNullCheckINS0_12ProjectEPCtx</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ZN7terrier11transaction18TransactionManager6CommitEPNS0_18TransactionEPNS</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ZN7terrier7storage11StorageUtil22CopyAttrIntoProjectionINSO_12ProjectedRkNSO</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ZN12Uniform_int_distributionInEcel1St26Linear_congruential_engine</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>_int_malloc</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>_sched_yield</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>cfree@GLIBC_2.2.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>_ZN11Function_handlerIfvvEZN7terrier31LargeTransactionBenchmark</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>_ZNK7terrier7storage9DataTable24AtomicallyReadVersionPtrENS0_9Tuple</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>_ZNK10_HashTableIN7terrier6common15StrongTypeAliasINS0_11TransactionEPNS0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>_ZN7terrier7storage16GarbageCollector18ProcessUnlinkQueueEv</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>_ZN7terrier7storage11StorageUtil17CopyWithNullCheckEPKSt4byteRKNSO</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>libtbb.so.2</td>
<td>@x00000000000018ac4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>libtbb.so.2</td>
<td>malloc</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>libtbb.so.2</td>
<td>_ZN11Function_handlerIfvjEZN7terrier31LargeTransactionBenchmark</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>libtbb.so.2</td>
<td>_memset_avx2_unaligned_erns</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>libtbb.so.2</td>
<td>_ZN11Function_handlerIfvvEZN7terrier31LargeTransactionBenchmark</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>libtbb.so.2</td>
<td>@x000000000000185cb</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>libtbb.so.2</td>
<td>_ZN7terrier31LargeTransactionBenchmarkObject22SimulateOneTransaction</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>libtbb.so.2</td>
<td>_Zst18generate_canonicalIdlm53Ext26Linear_congruential_engineInLm1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>libtbb.so.2</td>
<td>_ZN7terrier11transaction18TransactionManager9LogCommitEPNS0_18Transaction</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>libtbb.so.2</td>
<td>_ZN7terrier31LargeTransactionBenchmarkObject20PopulateInitialTable</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>libtbb.so.2</td>
<td>kernel</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>libtbb.so.2</td>
<td>kernel</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>libtbb.so.2</td>
<td>kernel</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>libtbb.so.2</td>
<td>kernel</td>
</tr>
</tbody>
</table>
```
PERF VISUALIZATION

We can also use `perf` to visualize the generated profile for our application.

$ perf report

There are also third-party visualization tools:
→ Hotspot
We can also use **perf** to visualize the generated profile for our application. There are also third-party visualization tools:
We can also use `perf` to visualize the generated profile for our application. There are also third-party visualization tools: → Hotspot.

```
perf report
```
PERF EVENTS

Supports several other events like:
→ L1-dcache-load-misses
→ branch-misses

To see a list of events:

```
$ perf list
```

Another usage example:

```
$ perf record -e cycles,LLC-load-misses -c 2000 ./relwithdebinfo/slot_iterator_benchmark
```
REFERENCES

Valgrind
→ The Valgrind Quick Start Guide
→ Callgrind
→ Kcacheegrind
→ Tips for the Profiling/Optimization process

Perf
→ Perf Tutorial
→ Perf Examples
→ Perf Analysis Tools
NEXT CLASS

Index Locking + Latching
T-Trees (1980s / TimesTen)
Bw-Tree (Hekaton)