Lecture #05

Carnegie Mellon University

ADVANCED DATABASE SYSTEMS

Multi-Version Concurrency Control (Garbage Collection)

@Andy_Pavlo // 15-721 // Spring 2020

MVCC GARBAGE COLLECTION

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

MVCC DELETES

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

SCALABLE GARBAGE COLLECTION FOR IN-MEMORY MVCC SYSTEMS
VLDB 2019

HYBRID GARBAGE COLLECTION FOR MULTI-VERSION CONCURRENCY CONTROL IN SAP HANA
SIGMOD 2016

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

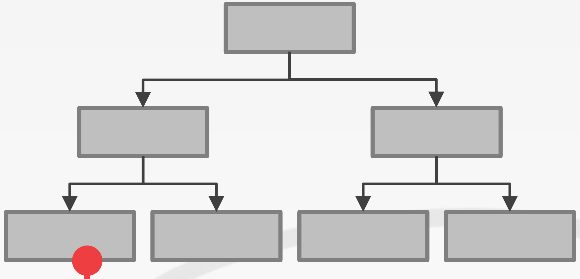
Thread #1

Begin @ 10

UPDATE(A)

key=222

Index



VERSION	BEGIN-TS	END-TS	KEY
A_1	1	∞	111

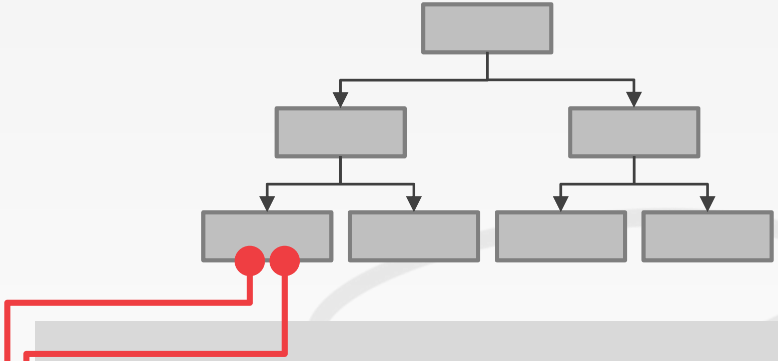
PELOTON MISTAKE

Thread #1

Begin @ 10

key=222

Index



VERSION	BEGIN-TS	END-TS	KEY
A_1	1	10	111
A_2	10	∞	222

PELOTON MISTAKE

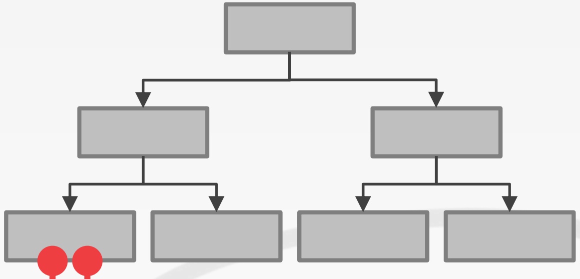
Thread #1

Begin @ 10

UPDATE(A)
key=222

UPDATE(A)
key=333

Index



VERSION	BEGIN-TS	END-TS	KEY
A_1	1	10	111
A_2	10	∞	222

PELOTON MISTAKE

Thread #1

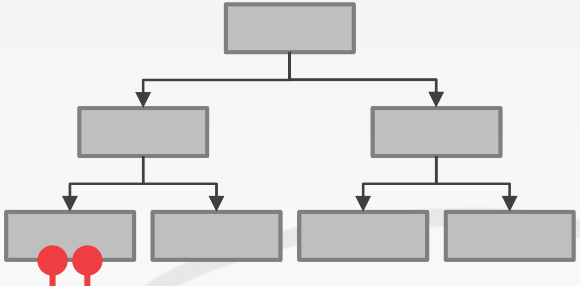
Begin @ 10

UPDATE(A)
key=222

UPDATE(A)
key=333

If a txn writes to same tuple more than once, then it just overwrites its previous version.

Index



VERSION	BEGIN-TS	END-TS	KEY
A_1	1	10	111
A_3	10	∞	333

PELOTON MISTAKE

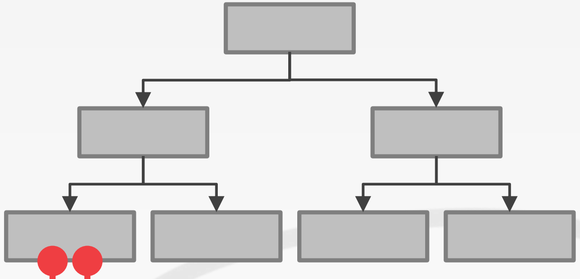
Thread #1

Begin @ 10

key=222 *key=333* *key=444*

If a txn writes to same tuple more than once, then it just overwrites its previous version.

Index



VERSION	BEGIN-TS	END-TS	KEY
A_1	1	10	111
A_4	10	∞	444

PELOTON MISTAKE

Thread #1

Begin @ 10

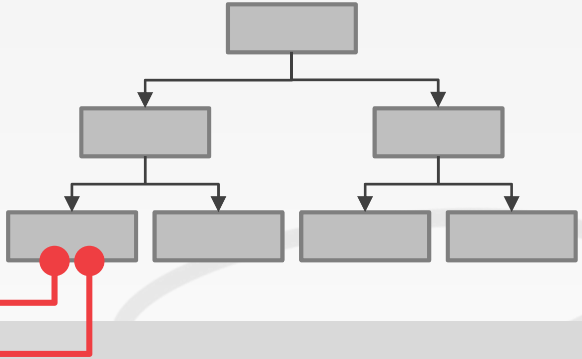
ABORT

UPDATE(A)	UPDATE(A)	UPDATE(A)
key=222	*key=333*	*key=444*

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.

Index



VERSION	BEGIN-TS	END-TS	KEY
A_1	1	10	111
A_4	10	∞	444

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

UPDATE(A)

	BEGIN-TS	END-TS	DATA
A_2	*1*	∞	-
B_6	*8*	∞	-

GC – VERSION TRACKING

Thread #1

Begin @ 10

UPDATE(A)

	BEGIN-TS	END-TS	DATA
A_2	1	10	-
B_6	8	∞	-
A_3	10	∞	-

GC – VERSION TRACKING

Thread #1

Begin @ 10

UPDATE(A)

Old Versions

A_2

	BEGIN-TS	END-TS	DATA
A_2	1	10	-
B_6	8	∞	-
A_3	10	∞	-

GC – VERSION TRACKING

Thread #1

Begin @ 10

Old Versions

A_2

UPDATE(A)

UPDATE(B)

	BEGIN-TS	END-TS	DATA
A_2	1	10	-
B_6	8	∞	-
A_3	10	∞	-

GC – VERSION TRACKING

Thread #1

Begin @ 10

Old Versions

A_2

UPDATE(A)

UPDATE(B)

	BEGIN-TS	END-TS	DATA
A_2	1	10	-
B_6	8	10	-
A_3	10	∞	-
B_7	10	∞	-

GC – VERSION TRACKING

Thread #1

Begin @ 10

Old Versions

A_2

B_6

UPDATE(A)

UPDATE(B)

	BEGIN-TS	END-TS	DATA
A_2	1	10	-
B_6	8	10	-
A_3	10	∞	-
B_7	10	∞	-

GC – VERSION TRACKING

Thread #1

Begin @ 10
Commit @ 15

Old Versions

A_2

B_6

UPDATE(A)

UPDATE(B)

	BEGIN-TS	END-TS	DATA
A_2	1	15	-
B_6	8	15	-
A_3	15	∞	-
B_7	15	∞	-

GC – VERSION TRACKING

Thread #1

Begin @ 10
Commit @ 15

Old Versions

UPDATE(A)

UPDATE(B)

	BEGIN-TS	END-TS	DATA
A_2	1	15	-
B_6	8	15	-
A_3	15	∞	-
B_7	15	∞	-

Vacuum

TS<15

A_2

B_6

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.

GC – GRANULARITY

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 reclamations.

GC – GRANULARITY

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.

GC – COMPARISON UNIT

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)

	BEGIN-TS	END-TS	DATA
A_1	1	25	-
A_2	25	35	-
A_3	35	∞	-

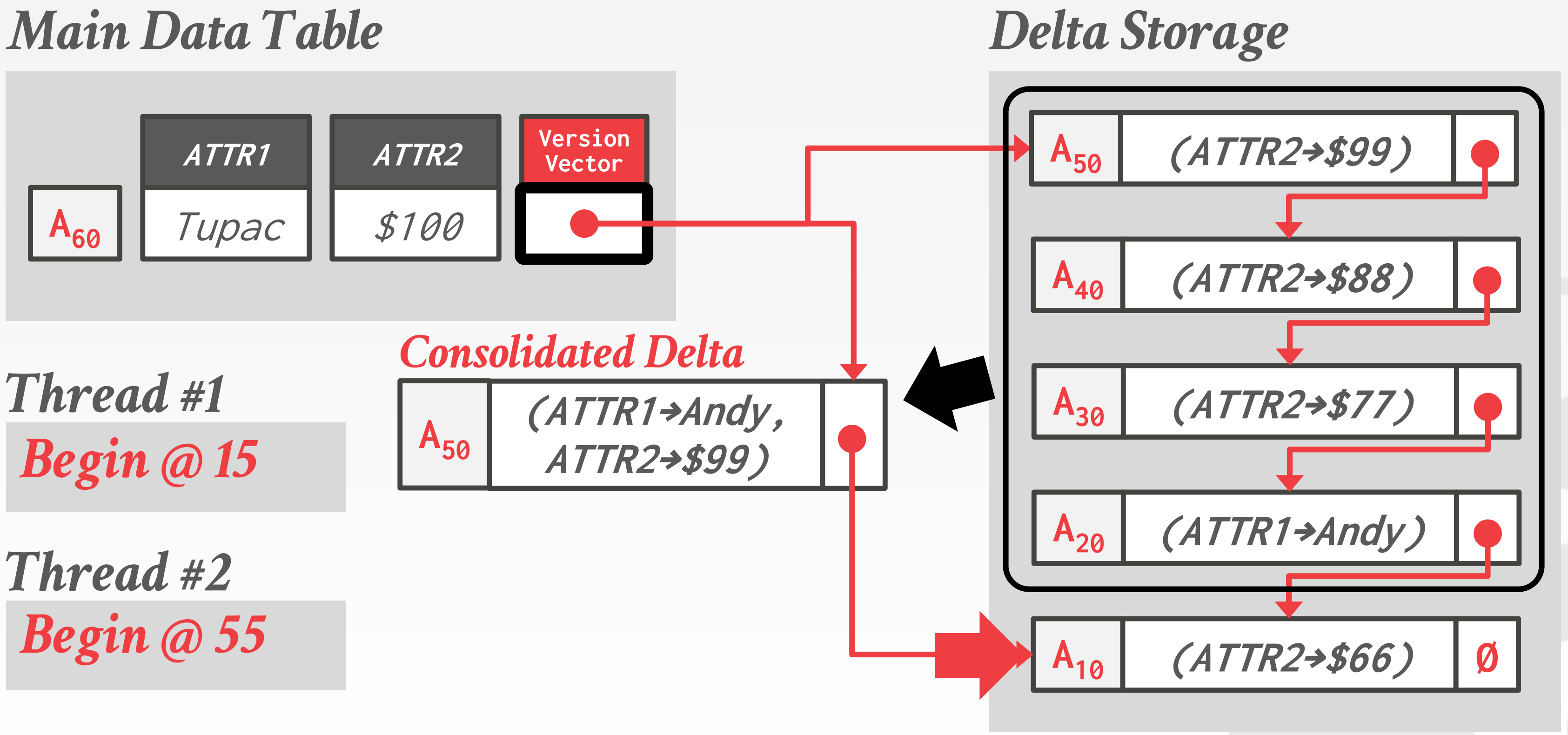
Timestamp

→ GC cannot reclaim A_2 because the lowest active txn TS (10) is less than END-TS.

Interval

→ GC can reclaim A_2 because no active txn TS intersects the interval [25,35].

GC – INTERVAL DELTA RECORDS



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.

BLOCK COMPACTION

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 COMPACTION – 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.

ANDY'S

TIPS FOR PROFILING

MOTIVATION

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

AMDAHL'S LAW

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.

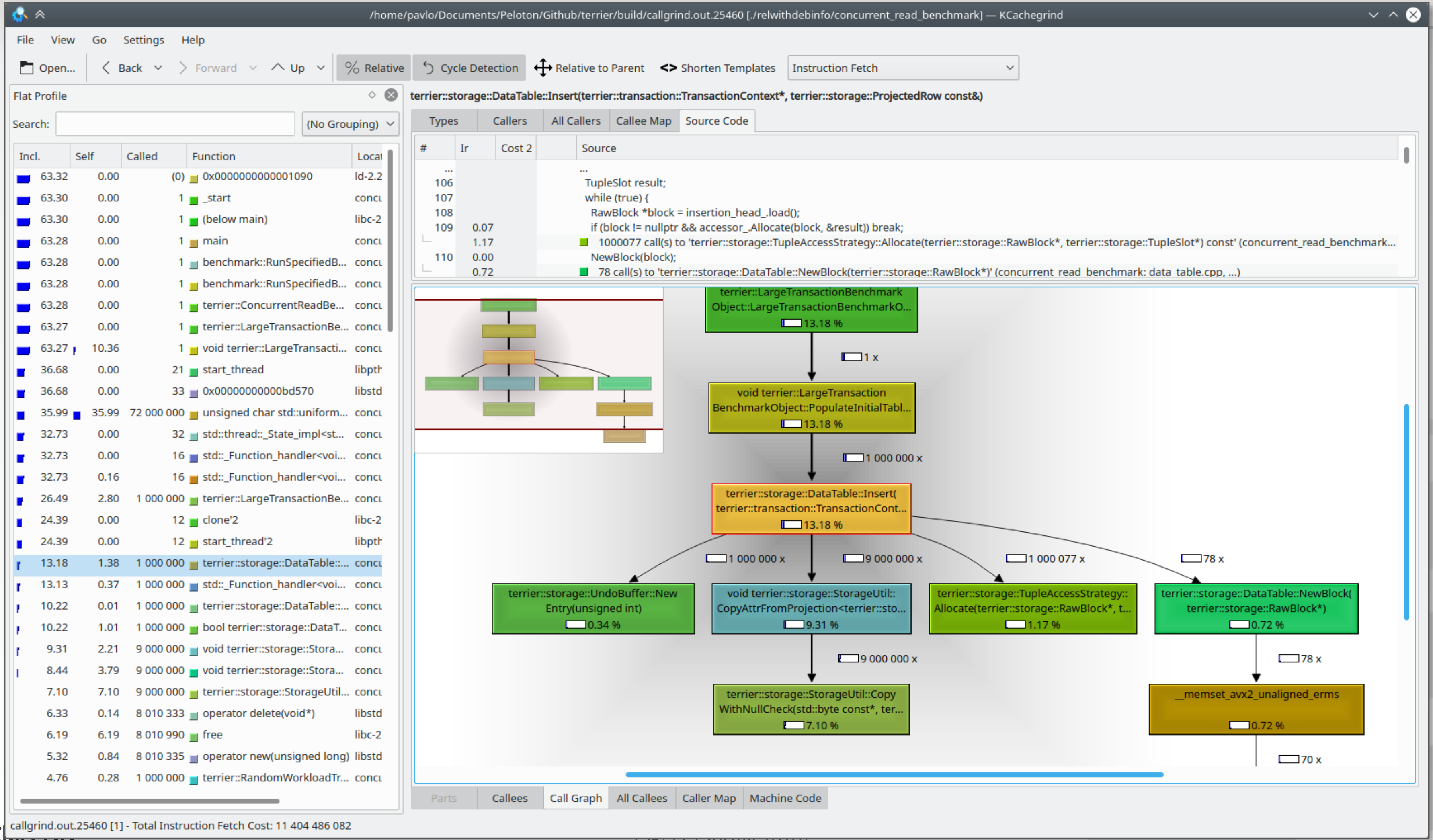
KCACHEGRIND

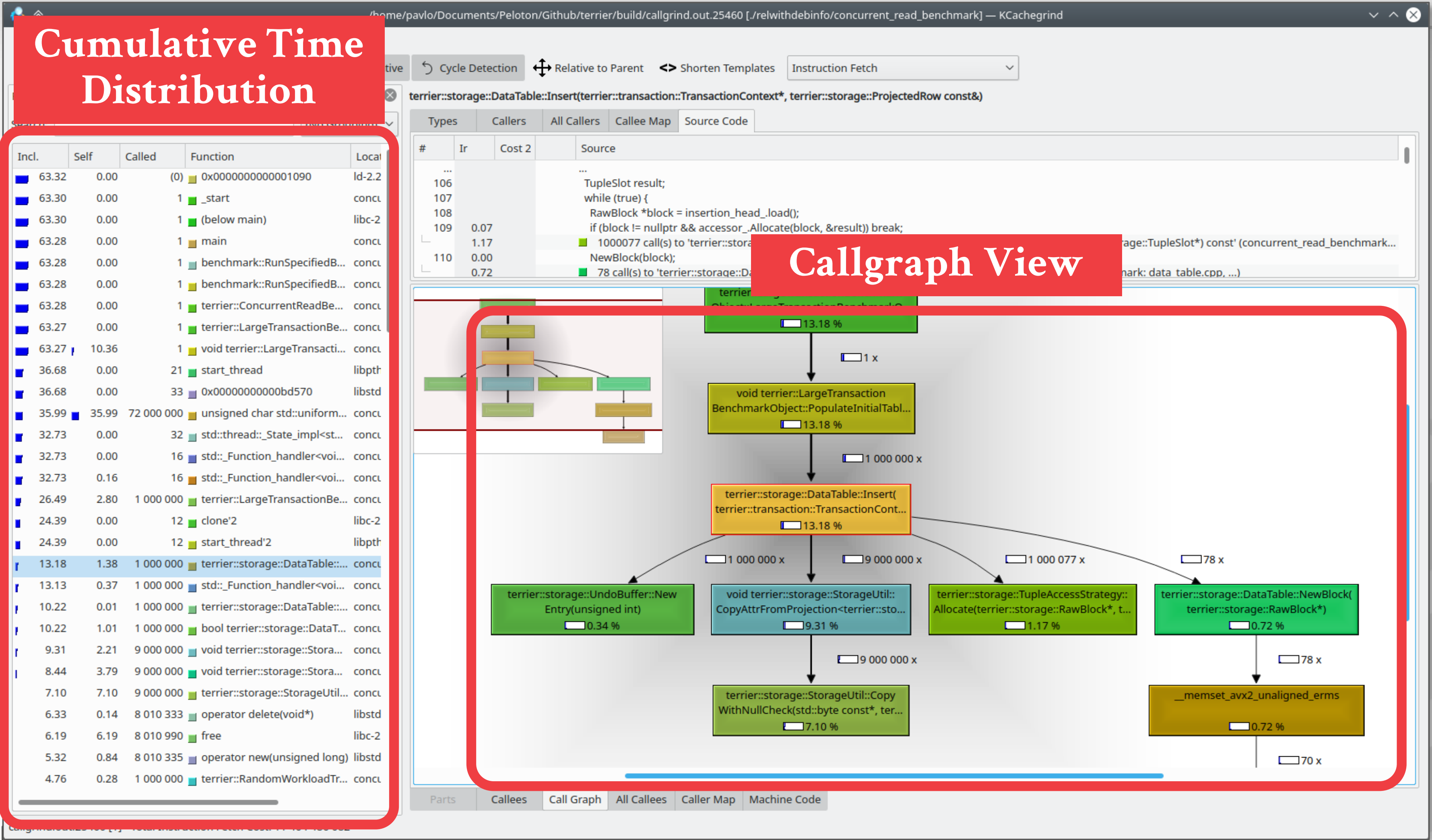
Using callgrind to profile the target benchmark and the overall DBMS in general:

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

Profile data visualization tool:

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


/home/pavlo/Documents/Peloton/Github/terrier/build/callgrind.out.25460 [./relwithdebinfo/concurrent_read_benchmark] — KCachegrind
File View Go Settings Help
Open... Back Forward Up Relative Cycle Detection Relative to Parent Shorten Templates Instruction Fetch
Flat Profile
Search: (No Grouping)
Incl. Self Called Function Locat
63.32 0.00 (0) 0x0000000000001090 ld-2.2
63.30 0.00 1 _start concu
63.30 0.00 1 (below main) libc-2
63.28 0.00 1 main concu
63.28 0.00 1 benchmark::RunSpecifiedB... concu
63.28 0.00 1 benchmark::RunSpecifiedB... concu
63.28 0.00 1 terrier::ConcurrentReadBe... concu
63.27 0.00 1 terrier::LargeTransactionBe... concu
63.27 10.36 1 void terrier::LargeTransacti... concu
36.68 0.00 21 start_thread libpth
36.68 0.00 33 0x00000000000bd570 libstd
35.99 35.99 72 000 000 unsigned char std::uniform... concu
32.73 0.00 32 std::thread::_State_impl<st... concu
32.73 0.00 16 std::_Function_handler<voi... concu
32.73 0.16 16 std::_Function_handler<voi... concu
26.49 2.80 1 000 000 terrier::LargeTransactionBe... concu
24.39 0.00 12 clone'2 libc-2
24.39 0.00 12 start_thread'2 libpth
13.18 1.38 1 000 000 terrier::storage::DataTable::... concu
13.13 0.37 1 000 000 std::_Function_handler<voi... concu
10.22 0.01 1 000 000 terrier::storage::DataTable::... concu
10.22 1.01 1 000 000 bool terrier::storage::DataT... concu
9.31 2.21 9 000 000 void terrier::storage::Stora... concu
8.44 3.79 9 000 000 void terrier::storage::Stora... concu
7.10 7.10 9 000 000 terrier::storage::StorageUtil... concu
6.33 0.14 8 010 333 operator delete(void*) libstd
6.19 6.19 8 010 990 free libc-2
5.32 0.84 8 010 335 operator new(unsigned long) libstd
4.76 0.28 1 000 000 terrier::RandomWorkloadTr... concu
terrier::storage::DataTable::Insert(terrier::transaction::TransactionContext*, terrier::storage::ProjectedRow const&)
Types Callers All Callers Callee Map Source Code
Ir Cost 2 Source
106 TupleSlot result;
107 while (true) {
108 RawBlock *block = insertion_head_.load();
109 0.07 if (block != nullptr && accessor_.Allocate(block, &result)) break;
1.17 1000077 call(s) to 'terrier::storage::TupleAccessStrategy::Allocate(terrier::storage::RawBlock*, terrier::storage::TupleSlot*) const' (concurrent_read_benchmark...
110 0.00 NewBlock(block);
0.72 78 call(s) to 'terrier::storage::DataTable::NewBlock(terrier::storage::RawBlock*)' (concurrent_read_benchmark: data_table.cpp, ...)
terrier::LargeTransactionBenchmark Object::LargeTransactionBenchmarkO... 13.18 %
1 x
void terrier::LargeTransaction BenchmarkObject::PopulateInitialTabl... 13.18 %
1 000 000 x
terrier::storage::DataTable::Insert(terrier::transaction::TransactionCont... 13.18 %
1 000 000 x
9 000 000 x
1 000 077 x
78 x
terrier::storage::UndoBuffer::New Entry(unsigned int) 0.34 %
void terrier::storage::StorageUtil:: CopyAttrFromProjection<terrier::sto... 9.31 %
terrier::storage::TupleAccessStrategy:: Allocate(terrier::storage::RawBlock*, t... 1.17 %
terrier::storage::DataTable::NewBlock(terrier::storage::RawBlock*) 0.72 %
9 000 000 x
78 x
terrier::storage::StorageUtil::Copy WithNullCheck(std::byte const*, ter... 7.10 %
__memset_avx2_unaligned_erms 0.72 %
70 x
Parts Callees Call Graph All Callees Caller Map Machine Code
callgrind.out.25460 [1] - Total Instruction Fetch Cost: 11 404 486 082


Cumulative Time Distribution
Callgraph View
/home/pavlo/Documents/Peloton/Github/terrier/build/callgrind.out.25460 [./relwithdebinfo/concurrent_read_benchmark] — KCachegrind
Cycle Detection
Relative to Parent
Shorten Templates
Instruction Fetch
terrier::storage::DataTable::Insert(terrier::transaction::TransactionContext*, terrier::storage::ProjectedRow const&)
Types
Callers
All Callers
Callee Map
Source Code
Incl.
Self
Called
Function
TupleSlot result;
while (true) {
RawBlock *block = insertion_head_.load();
if (block != nullptr && accessor_.Allocate(block, &result)) break;
NewBlock(block);
void terrier::LargeTransaction BenchmarkObject::PopulateInitialTabl...
13.18 %
1 000 000 x
terrier::storage::DataTable::Insert(terrier::transaction::TransactionCont...
13.18 %
terrier::storage::UndoBuffer::New Entry(unsigned int)
0.34 %
void terrier::storage::StorageUtil:: CopyAttrFromProjection<terrier::sto...
9.31 %
terrier::storage::TupleAccessStrategy:: Allocate(terrier::storage::RawBlock*, t...
1.17 %
terrier::storage::DataTable::NewBlock(terrier::storage::RawBlock*)
0.72 %
9 000 000 x
1 000 077 x
78 x
terrier::storage::StorageUtil::Copy WithNullCheck(std::byte const*, ter...
7.10 %
__memset_avx2_unaligned_erms
0.72 %
70 x
Parts
Callees
Call Graph
All Callees
Caller Map
Machine Code

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

```
Samples: 9M of event 'cycles:u', Event count (approx.): 18388130000
Overhead  Command          Shared Object              Symbol
  17.89%  concurrent_read  concurrent_read_benchmark  [.] _ZN7terrier7storage11StorageUtil17CopyWithNullCheckINS0_12Projecte
   9.41%  concurrent_read  concurrent_read_benchmark  [.] _ZN7terrier11transaction18TransactionManager6CommitEPNS0_18Transac
   9.36%  concurrent_read  concurrent_read_benchmark  [.] _ZN7terrier11transaction18TransactionManager16BeginTransactionEPNS
   8.10%  concurrent_read  concurrent_read_benchmark  [.] _ZNSt24uniform_int_distributionIhEclISt26linear_congruential_engin
   5.27%  concurrent_read  concurrent_read_benchmark  [.] _ZN7terrier7storage11StorageUtil22CopyAttrIntoProjectionINS0_12Pro
   3.53%  concurrent_read  libc-2.27.so               [.] _int_malloc
   3.28%  concurrent_read  libc-2.27.so               [.] __sched_yield
   3.08%  concurrent_read  libc-2.27.so               [.] cfree@GLIBC_2.2.5
   3.06%  concurrent_read  concurrent_read_benchmark  [.] _ZNSt17_Function_handlerIFvvEZN7terrier31LargeTransactionBenchmark
   2.87%  concurrent_read  concurrent_read_benchmark  [.] _ZNK7terrier7storage9DataTable24AtomicallyReadVersionPtrENS0_9Tupl
   2.72%  concurrent_read  concurrent_read_benchmark  [.] _ZNKSt10_HashtableIN7terrier6common15StrongTypeAliasINS0_11transac
   2.45%  concurrent_read  concurrent_read_benchmark  [.] _ZN7terrier7storage16GarbageCollector18ProcessUnlinkQueueEv
   1.86%  concurrent_read  concurrent_read_benchmark  [.] _ZN7terrier7storage11StorageUtil17CopyWithNullCheckEPKSt4byteRKNS0
   1.74%  concurrent_read  libtbb.so.2                [.] 0x0000000000018ac4
   1.58%  concurrent_read  libc-2.27.so               [.] malloc
   1.20%  concurrent_read  concurrent_read_benchmark  [.] _ZNSt10_HashtableIN7terrier6common15StrongTypeAliasINS0_11transact
   0.99%  concurrent_read  libc-2.27.so               [.] __memset_avx2_unaligned_erms
   0.98%  concurrent_read  concurrent_read_benchmark  [.] _ZNSt17_Function_handlerIFvjEZN7terrier31LargeTransactionBenchmark
   0.90%  concurrent_read  libtbb.so.2                [.] 0x00000000000185cb
   0.89%  concurrent_read  concurrent_read_benchmark  [.] _ZN7terrier31LargeTransactionBenchmarkObject22SimulateOneTransacti
   0.83%  concurrent_read  concurrent_read_benchmark  [.] _ZSt18generate_canonicalIdLm53ESt26linear_congruential_engineImLm1
   0.72%  concurrent_read  concurrent_read_benchmark  [.] _ZN7terrier11transaction18TransactionManager9LogCommitEPNS0_18Tran
                            ncurrent_read_benchmark  [.] _ZN7terrier31LargeTransactionBenchmarkObject20PopulateInitialTable
                            btbb.so.2                [.] 0x0000000000018ac6
                            ernel]                   [k] 0xfffffe000005e000
                            ncurrent_read_benchmark  [.] _ZNK7terrier7storage9DataTable16SelectIntoBufferINS0_12ProjectedRo
                            ernel]                   [k] 0xfffffe00000e2000
Cannot load tips.txt file, please install perf!
```

Cumulative Event Distribution

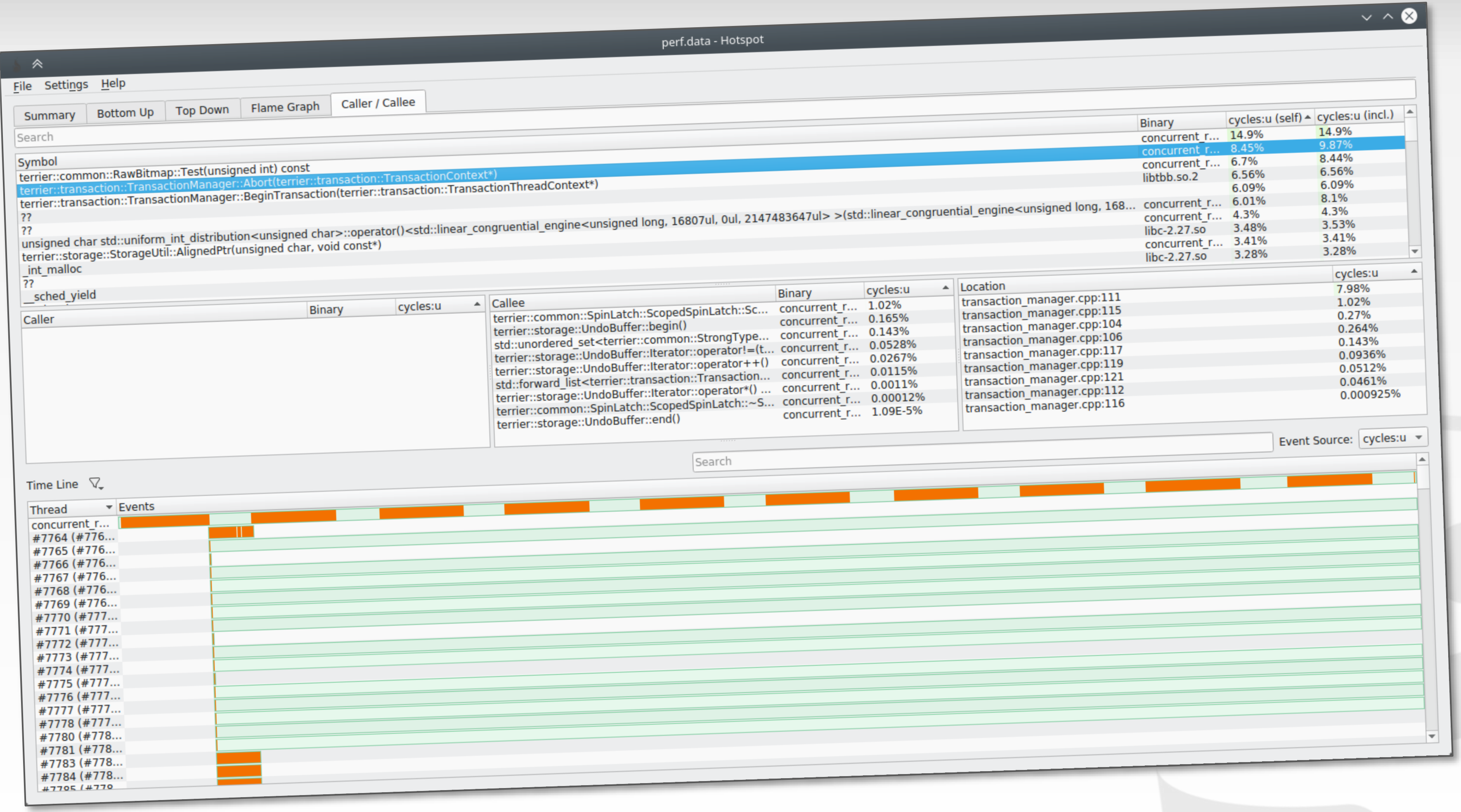
PERF VISUALIZATION

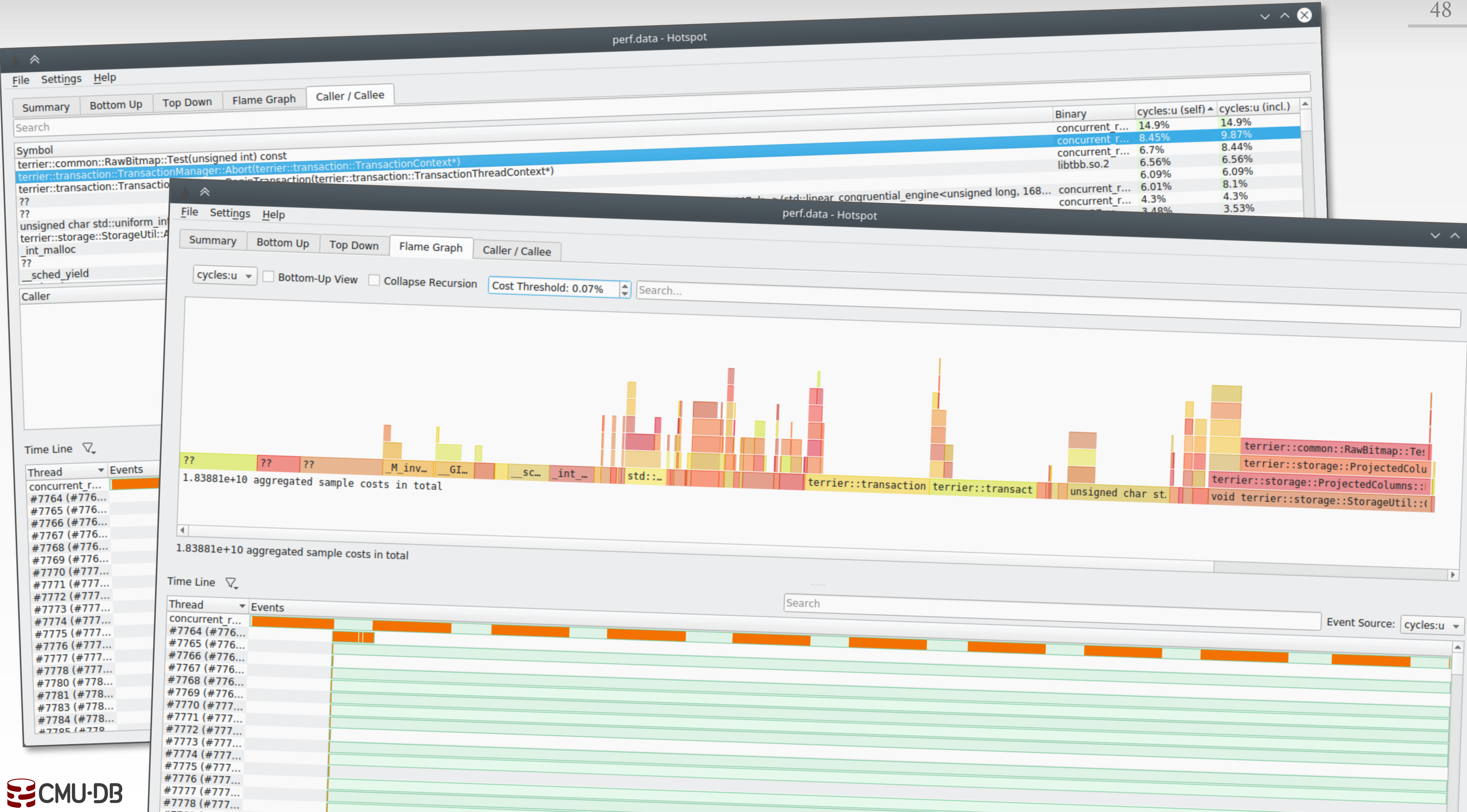
We can also use **perf** to visualize the generated profile for our application.

```
$ perf report
```

There are also third-party visualization tools:

→ Hotspot


perf.data - Hotspot
File Settings Help
Summary Bottom Up Top Down Flame Graph Caller / Callee
Search
Symbol
terrier::common::RawBitmap::Test(unsigned int) const
terrier::transaction::TransactionManager::Abort(terrier::transaction::TransactionContext*)
terrier::transaction::TransactionManager::BeginTransaction(terrier::transaction::TransactionThreadContext*)
??
??
unsigned char std::uniform_int_distribution<unsigned char>::operator()<std::linear_congruential_engine<unsigned long, 16807ul, 0ul, 2147483647ul> >(std::linear_congruential_engine<unsigned long, 168...
terrier::storage::StorageUtil::AlignedPtr(unsigned char, void const*)
_int_malloc
??
__sched_yield
Binary cycles:u (self) cycles:u (incl.)
concurrent_r... 14.9% 14.9%
concurrent_r... 8.45% 9.87%
concurrent_r... 6.7% 8.44%
libtbb.so.2 6.56% 6.56%
6.09% 6.09%
concurrent_r... 6.01% 8.1%
concurrent_r... 4.3% 4.3%
libc-2.27.so 3.48% 3.53%
concurrent_r... 3.41% 3.41%
libc-2.27.so 3.28% 3.28%
Caller Binary cycles:u
Callee Binary cycles:u
terrier::common::SpinLatch::ScopedSpinLatch::Sc... concurrent_r... 1.02%
terrier::storage::UndoBuffer::begin() concurrent_r... 0.165%
std::unordered_set<terrier::common::StrongType... concurrent_r... 0.143%
terrier::storage::UndoBuffer::Iterator::operator!=(t... concurrent_r... 0.0528%
terrier::storage::UndoBuffer::Iterator::operator++() concurrent_r... 0.0267%
std::forward_list<terrier::transaction::Transaction... concurrent_r... 0.0115%
terrier::storage::UndoBuffer::Iterator::operator*() ... concurrent_r... 0.0011%
terrier::common::SpinLatch::ScopedSpinLatch::~S... concurrent_r... 0.00012%
terrier::storage::UndoBuffer::end() concurrent_r... 1.09E-5%
Location cycles:u
transaction_manager.cpp:111 7.98%
transaction_manager.cpp:115 1.02%
transaction_manager.cpp:104 0.27%
transaction_manager.cpp:106 0.264%
transaction_manager.cpp:117 0.143%
transaction_manager.cpp:119 0.0936%
transaction_manager.cpp:121 0.0512%
transaction_manager.cpp:112 0.0461%
transaction_manager.cpp:116 0.000925%
Event Source: cycles:u
Search
Time Line
Thread Events
concurrent_r...
#7764 (#776...
#7765 (#776...
#7766 (#776...
#7767 (#776...
#7768 (#776...
#7769 (#776...
#7770 (#777...
#7771 (#777...
#7772 (#777...
#7773 (#777...
#7774 (#777...
#7775 (#777...
#7776 (#777...
#7777 (#777...
#7778 (#777...
#7780 (#778...
#7781 (#778...
#7783 (#778...
#7784 (#778...


perf.data - Hotspot
File Settings Help
Summary
Bottom Up
Top Down
Flame Graph
Caller / Callee
Search
Symbol
terrier::common::RawBitmap::Test(unsigned int) const
terrier::transaction::TransactionManager::Abort(terrier::transaction::TransactionContext*)
BeginTransaction(terrier::transaction::TransactionThreadContext*)
unsigned char std::uniform_int
terrier::storage::StorageUtil::
_int_malloc
__sched_yield
Binary
cycles:u (self)
cycles:u (incl.)
concurrent_r... 14.9% 14.9%
concurrent_r... 8.45% 9.87%
concurrent_r... 6.7% 8.44%
libtbb.so.2 6.56% 6.56%
6.09% 6.09%
concurrent_r... 6.01% 8.1%
concurrent_r... 4.3% 4.3%
3.53%
Caller
Time Line
Thread
Events
perf.data - Hotspot
File Settings Help
Summary
Bottom Up
Top Down
Flame Graph
Caller / Callee
cycles:u
Bottom-Up View
Collapse Recursion
Cost Threshold: 0.07%
Search...
terrier::common::RawBitmap::Tes
terrier::storage::ProjectedColu
terrier::storage::ProjectedColumns::
void terrier::storage::StorageUtil::
terrier::transaction
terrier::transact
unsigned char st
1.83881e+10 aggregated sample costs in total
1.83881e+10 aggregated sample costs in total
Time Line
Thread
Events
Search
Event Source:
cycles:u
concurrent_r...
#7764 (#776...
#7765 (#776...
#7766 (#776...
#7767 (#776...
#7768 (#776...
#7769 (#776...
#7770 (#777...
#7771 (#777...
#7772 (#777...
#7773 (#777...
#7774 (#777...
#7775 (#777...
#7776 (#777...
#7777 (#777...
#7778 (#777...

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
→ Kcachegrind
→ 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)