Performing Low-Level I/O Evaluations for Discovering Potential I/O Issues using IOscope

Abdulqawi Saif¹,²
abdulqawi.saif@loria.fr

Lucas Nussbaum¹
lucas.nussbaum@loria.fr

Ye-Qiong Song¹
ye-qiong.song@loria.fr

¹Université de Lorraine, CNRS, Inria, LORIA, F-54000 Nancy, France
²Qwant Enterprise, F-88000 Épinal, France

January 25, 2019
Per3S 2019 – Performance and Scalability of Storage Systems
Bordeaux, France
How is I/O performance often evaluated?

Storage system X
How is I/O performance often evaluated?

⇒ Communication through high-level interfaces (REST, API, ...)

Storage system X

Test eng.
How is I/O performance often evaluated?

⇒ Workload execution phase
⇒ Configuration determines which nodes to use
How is I/O performance often evaluated?

⇒ High-level metrics (e.g., overall I/O throughput, exec. time, ...)
⇒ Results: aggregated and reported by the storage system
How is I/O performance often evaluated?

⇒ Lack of I/O analysis tools (measurements ≠ understanding)
⇒ Potential I/O errors in lower layers are ignored
⇒ Nothing is known about workloads’ data access
  → Pattern-related errors?
Context

**Goal**

⇒ Analyzing I/O patterns\(^1\) of storage workloads

**Requirements**

⇒ Flexible and simple as high-level evaluations
⇒ straightforward results
⇒ Work in production environments
  → Negligible overhead
  → Verified behaviours in lower layers

---

\(^1\) We define I/O access pattern of a given workload as the sequences of the I/O requests issued by the target I/O process during a given workload to access on-disk data files
Context

Tracing in the I/O evaluation context

IOscope\(^1\) design & validation

Experiments on MongoDB & Cassandra

Conclusions

\(^1\) https://github.com/LeUnAiDeS/IOscope (reproducible scenarios provided)
Tracing in the I/O evaluation context

⇒ Generic tracing – no filtering mechanism[1]
  → Heavy tracing output
  → Post-analysis effort is not negligible

Tracing in the I/O evaluation context

⇒ **Generic tracing – no filtering mechanism**[1]
  → Heavy tracing output
  → Post-analysis effort is not negligible

⇒ **Support partial storage media**[2]
  → Works only with HDDs

Tracing in the I/O evaluation context

⇒ Generic tracing – no filtering mechanism\[1\]
  → Heavy tracing output
  → Post-analysis effort is not negligible

⇒ Support partial storage media\[2\]
  → Works only with HDDs

⇒ Partial coverage of filesystems\[3\]
Tracing in the I/O evaluation context

⇒ Generic tracing – no filtering mechanism[1]
  → Heavy tracing output
  → Post-analysis effort is not negligible
⇒ Support partial storage media[2]
  → Works only with HDDs
⇒ Partial coverage of filesystems[3]

Tracing in the I/O evaluation context

⇒ **Generic tracing – no filtering mechanism[1]**
   → Heavy tracing output
   → Post-analysis effort is not negligible

⇒ **Support partial storage media[2]**
   → Works only with HDDs

⇒ **Partial coverage of filesystems[3]**

⇒ **Modified kernel is required[3]**

Tracing in the I/O evaluation context

⇒ Generic tracing – no filtering mechanism[1]
  → Heavy tracing output
  → Post-analysis effort is not negligible
⇒ Support partial storage media[2]
  → Works only with HDDs
⇒ Partial coverage of filesystems[3]
⇒ Modified kernel is required[3]
⇒ Many tracing frameworks for I/O extrapolation[4,5,6,7,8]
  → Different scope – traces are more important than workloads!

Tracing in the I/O evaluation context

⇒ Generic tracing – no filtering mechanism[1]
  → Heavy tracing output
  → Post-analysis effort is not negligible
⇒ Support partial storage media[2]
  → Works only with HDDs
⇒ Partial coverage of filesystems[3]
⇒ Modified kernel is required[3]
⇒ Many tracing frameworks for I/O extrapolation[4,5,6,7,8]
  → Different scope – traces are more important than workloads!
⇒ Tools like DTrace, SystemTap, LTTng use dynamic loading
  → Usage suitability is affected (e.g., signed kernels, no compilation tools)
Tracing in the I/O evaluation context

- Generic tracing – no filtering mechanism [1]
  - Heavy tracing output
  - Post-analysis effort is not negligible
- Support partial storage media [2]
  - Works only with HDDs
- Partial coverage of filesystems [3]
- Modified kernel is required [3]
- Many tracing frameworks for I/O extrapolation [4, 5, 6, 7, 8]
  - Different scope – traces are more important than workloads!
- Tools like DTrace, SystemTap, LTTng use dynamic loading
  - Usage suitability is affected (e.g., signed kernels, no compilation tools)
- Partial coverage of I/O methods such as mmap [9, 10, 11]

[10] IOVISOR BCC Project’s slower tools (filesize, ext4slo).  
IOscope is based on eBPF. What is eBPF?

⇒ A recent tracing and filtering technology
⇒ Connect to all data sources: (Kprobes, Uprobes, tracepoints, ...)
⇒ Almost near-zero overhead (4 ns per syscall)$^2$
⇒ Formally adopted by the Linux kernel ($\geq$ Linux 3.19)
⇒ Has a lot of front-end projects like IOVISOR‘s BCC
   → No more byte code!
   → Towards precise-objective tracing

**IOscope design & validation**

**IOscope tracer: uncovering I/O patterns for storage workloads**

- **filtering-based tracing** mechanism
  - Reduce the collected data by an order of magnitude
  - Less interceptions = lower overhead \( (\leq 0.08\%) \)
  - Tiny tracing granularities (e.g. R/W operations)

- Two tools for:
  - I/O workloads issued using **syscalls**
  - memory **mapped-files** workloads

- Collect specific data \{file offsets, size, latency, timestamps, op.Mode\}

- Useful for in-production usage

---

3. Several filters are applied in both kernel and userspace pair during collecting traces (pid, files, I/O operations, ...)

---

Performing Low-Level I/O Evaluations for Discovering Potential I/O Issues using IOscope
**IOscope validation**

- Linux kernel 4.9.0
- Flexible Input/Output (FIO) benchmark is used
  - Generate diverse workloads / several I/O methods
    - *E.g. rand reading workload for mmap:*
      ```
      fio -name=testfile -rw=randread -ioengine=mmap -direct=0 -size=10G -numjobs=1 -group_reporting
      ```

**Table – Validated I/O access modes and workloads**

<table>
<thead>
<tr>
<th>Fio IOengine</th>
<th>Target syscalls</th>
<th>Tested workloads: read, write, randread, randwrite, readwrite, and randreadwrite</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sync</td>
<td>read, write</td>
<td>all</td>
</tr>
<tr>
<td>Psync</td>
<td>pread, pwrite</td>
<td>all</td>
</tr>
<tr>
<td>Pvsync</td>
<td>preadv, pwritev</td>
<td>all</td>
</tr>
<tr>
<td>Pvsync2</td>
<td>preadv2, pwritev2</td>
<td>all</td>
</tr>
<tr>
<td>posixaio</td>
<td>aio_read, aio_write</td>
<td>all</td>
</tr>
<tr>
<td>Mmap</td>
<td>mmap, memcpy</td>
<td>all</td>
</tr>
</tbody>
</table>
IOscope validation

- Linux kernel 4.9.0
- Flexible Input/Output (FIO) benchmark is used
  - Generate diverse workloads / several I/O methods
  - E.g. rand reading workload for mmap:
    fio -name=testfile -rw=randread
    -ioengine=mmap -direct=0 -size=10G
    -numjobs=1 -group_reporting

Some results of validated workloads:

- rw workload - Poxisaio IOengine
- randrw workload - Psync IOengine
- r workload - mmap IOengine
Objective: Uncovering potential pattern-related issues

Experimental setup

- MongoDB & Cassandra are tested
  - MongoDB v3.4 with WiredTiger (classic I/O)
  - Cassandra v3.0.14 (mmap I/O)
- One client to index a simple int field
- Single server & two-shards cluster configurations
- Hash sharding for clustered configuration – load balancing
- Experiments run on HDDs & SSDs separately
- Cache is cleaned between experiments
- Data contiguity is tested using FIBMAP

Datasets

- Two equally-sized datasets (same characteristics)

<table>
<thead>
<tr>
<th>(min, avrg, max) in KB</th>
<th>N. of data units</th>
<th>Size (Gb)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1, 3.47, 6)</td>
<td>20,000,000 docs</td>
<td>71</td>
</tr>
</tbody>
</table>

- Elements x (int, date, 2 x str[min, max], array[1..4] x string[min, max])
I/O throughput of a single server experiment (HDD)

I/O throughput of Cassandra’s two-nodes cluster (HDD)
Experimentation - Cassandra results

→ Clustered experiments also have pure sequential access pattern
Experimentation - MongoDB results

Single-server experiment – HDD

Single-server experiment – SSD
Experimentation - MongoDB results

**Figure** – Results on HDD. Single-server & two different runs of distributed experiments

**Figure** – Results on SSD. Single-server & two different runs of distributed experiments
Experimentation - MongoDB results

Single-server I/O patterns: HDD (left) and SSD (right)
Experimentation - MongoDB results

Single-server I/O patterns: HDD (left) and SSD (right)

⇒ Max overhead is less than 0.80%
⇒ Acceptable access pattern on single-server experiments
⇒ Seq. access → Random access on shards!
⇒ Data distribution issue
⇒ SSDs are affected by I/O patterns too!
The exact issue

⇒ Mismatch between the scanning table vs data stored on disk

Collection _ids:
used by MongoDB process and WiredTiger

<table>
<thead>
<tr>
<th>_id 1</th>
<th>_id 2</th>
<th>_id 3</th>
<th>_id 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>_id 7</td>
<td>_id 8</td>
<td>_id 10</td>
<td>_id 11</td>
</tr>
<tr>
<td>_id 19</td>
<td>_id 20</td>
<td>_id 27</td>
<td>...</td>
</tr>
</tbody>
</table>

Collection file on disk:
allocated regarding the key sharding (hashed _id)

<table>
<thead>
<tr>
<th>_id 1</th>
<th>_id 3</th>
<th>_id 7</th>
<th>_id 5</th>
<th>_id 8</th>
<th>_id 10</th>
<th>_id 2</th>
<th>_id 19</th>
<th>_id 11</th>
<th>_id 20</th>
<th>_id 30</th>
<th>_id 27</th>
</tr>
</thead>
<tbody>
<tr>
<td>rec 1</td>
<td>rec 2</td>
<td>rec 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>rec(n-1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>rec(n)</td>
<td></td>
</tr>
</tbody>
</table>

Figure – MongoDB scanning table Vs records’ order on the disk
**The exact issue**

⇒ Mismatch between the scanning table vs data stored on disk

Collection _ids:
used by MongoDB process and WiredTiger

<table>
<thead>
<tr>
<th>_id 1</th>
<th>_id 2</th>
<th>_id 3</th>
<th>_id 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>_id 7</td>
<td>_id 8</td>
<td>_id 10</td>
<td>_id 11</td>
</tr>
<tr>
<td>_id 19</td>
<td>_id 20</td>
<td>_id 27</td>
<td>...</td>
</tr>
</tbody>
</table>

1) Get this doc

Collection file on disk:
allocated regarding the key sharding (hashed _id)

<table>
<thead>
<tr>
<th>_id 1</th>
<th>_id 3</th>
<th>_id 7</th>
<th>_id 5</th>
<th>_id 8</th>
<th>_id 10</th>
<th>_id 2</th>
<th>_id 19</th>
<th>_id 11</th>
<th>_id 20</th>
<th>_id 30</th>
<th>_id 27</th>
</tr>
</thead>
<tbody>
<tr>
<td>rec 1</td>
<td>rec 2</td>
<td>rec 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>rec(n-1)</td>
<td></td>
</tr>
</tbody>
</table>

**Figure** — MongoDB scanning table Vs records’ order on the disk
Experimentation - MongoDB results

The exact issue
⇒ Mismatch between the scanning table vs data stored on disk

Collection _ids:
used by MongoDB process and WiredTiger

Collection file on disk:
allocated regarding the key sharding (hashed _id)

1) Get this doc

Figure – MongoDB scanning table Vs records’ order on the disk
The exact issue
⇒ Mismatch between the scanning table vs data stored on disk

Collection _ids:
used by MongoDB process and WiredTiger

<table>
<thead>
<tr>
<th>_id 1</th>
<th>_id 2</th>
<th>_id 3</th>
<th>_id 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>_id 7</td>
<td>_id 8</td>
<td>_id 10</td>
<td>_id 11</td>
</tr>
<tr>
<td>_id 19</td>
<td>_id 20</td>
<td>_id 27</td>
<td>...</td>
</tr>
</tbody>
</table>

Collection file on disk:
allocated regarding the key sharding (hashed _id)

<table>
<thead>
<tr>
<th>_id 1</th>
<th>_id 3</th>
<th>_id 7</th>
<th>_id 5</th>
<th>_id 8</th>
<th>_id 10</th>
<th>_id 2</th>
<th>_id 19</th>
<th>_id 11</th>
<th>_id 20</th>
<th>_id 30</th>
<th>_id 27</th>
</tr>
</thead>
<tbody>
<tr>
<td>rec 1</td>
<td>rec 2</td>
<td>rec 3</td>
<td>rec 5</td>
<td>rec 8</td>
<td>rec 10</td>
<td>rec 2</td>
<td>rec 19</td>
<td>rec 11</td>
<td>rec 20</td>
<td>rec 30</td>
<td>rec (n)</td>
</tr>
</tbody>
</table>

1) Get this doc
2) Which doc is next?

**Figure** – MongoDB scanning table Vs records’ order on the disk
The exact issue
⇒ Mismatch between the scanning table vs data stored on disk

Collection _ids:
used by MongoDB process and WiredTiger

<table>
<thead>
<tr>
<th>_id 1</th>
<th>_id 2</th>
<th>_id 3</th>
<th>_id 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>_id 7</td>
<td>_id 8</td>
<td>_id 10</td>
<td>_id 11</td>
</tr>
<tr>
<td>_id 19</td>
<td>_id 20</td>
<td>_id 27</td>
<td>...</td>
</tr>
</tbody>
</table>

1) Get this doc
2) Which doc is next?

Collection file on disk:
allocated regarding the key sharding (hashed _id)

<table>
<thead>
<tr>
<th>_id 1</th>
<th>_id 3</th>
<th>_id 7</th>
<th>_id 5</th>
<th>_id 8</th>
<th>_id 10</th>
<th>_id 2</th>
<th>_id 19</th>
<th>_id 11</th>
<th>_id 20</th>
<th>_id 30</th>
<th>_id 27</th>
</tr>
</thead>
<tbody>
<tr>
<td>rec 1</td>
<td>rec 2</td>
<td>rec 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>rec(n-1)</td>
<td>rec(n)</td>
</tr>
</tbody>
</table>

**Figure** – MongoDB scanning table Vs records’ order on the disk
Experimentation - MongoDB results

The exact issue

⇒ Mismatch between the scanning table vs data stored on disk

Collection _ids:
used by MongoDB process and WiredTiger

| _id 1 | _id 2 | _id 3 | _id 5 |
| _id 7 | _id 8 | _id 10 | _id 11 |
| _id 19 | _id 20 | _id 27 | ... |

Collection file on disk:
allocated regarding the key sharding (hashed _id)

| _id 1 | _id 3 | _id 7 | _id 5 | _id 8 | _id 10 | _id 2 | _id 19 | _id 11 | _id 20 | _id 30 | _id 27 |
| rec 1 | rec 2 | rec 3 | rec  | rec 5 | rec 10 | rec 2 | rec 19 | rec 11 | rec 20 | rec 30 | rec (n) |

Figure – MongoDB scanning table Vs records’ order on the disk

1) Get this doc
The exact issue

⇒ Mismatch between the scanning table vs data stored on disk

Collection _ids:
used by MongoDB process and WiredTiger

<table>
<thead>
<tr>
<th>_id 1</th>
<th>_id 2</th>
<th>_id 3</th>
<th>_id 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>_id 7</td>
<td>_id 8</td>
<td>_id 10</td>
<td>_id 11</td>
</tr>
<tr>
<td>_id 19</td>
<td>_id 20</td>
<td>_id 27</td>
<td></td>
</tr>
</tbody>
</table>

1) Get this doc
2) Which doc is next?

Collection file on disk:
allocated regarding the key sharding (hashed _id)

<table>
<thead>
<tr>
<th>_id 1</th>
<th>_id 3</th>
<th>_id 7</th>
<th>_id 5</th>
<th>_id 8</th>
<th>_id 10</th>
<th>_id 2</th>
<th>_id 19</th>
<th>_id 11</th>
<th>_id 20</th>
<th>_id 30</th>
<th>_id 27</th>
</tr>
</thead>
<tbody>
<tr>
<td>rec 1</td>
<td>rec 2</td>
<td>rec 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>rec(n-1)</td>
<td>rec (n)</td>
</tr>
</tbody>
</table>

**Figure** – MongoDB scanning table Vs records’ order on the disk
The exact issue
⇒ Mismatch between the scanning table vs data stored on disk

Figure – MongoDB scanning table Vs records’ order on the disk
The exact issue

⇒ Mismatch between the scanning table vs data stored on disk

Collection _ids:
used by MongoDB process and WiredTiger

<table>
<thead>
<tr>
<th></th>
<th>_id 1</th>
<th>_id 2</th>
<th>_id 3</th>
<th>_id 5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>_id 7</td>
<td>_id 8</td>
<td>_id 10</td>
<td>_id 11</td>
</tr>
<tr>
<td></td>
<td>_id 19</td>
<td>_id 20</td>
<td>_id 27</td>
<td>...</td>
</tr>
</tbody>
</table>

1) Get this doc
2) Which doc is next?

Collection file on disk:
allocated regarding the key sharding (hashed _id)

<table>
<thead>
<tr>
<th></th>
<th>_id 1</th>
<th>_id 3</th>
<th>_id 7</th>
<th>_id 5</th>
<th>_id 8</th>
<th>_id 10</th>
<th>_id 2</th>
<th>_id 19</th>
<th>_id 11</th>
<th>_id 20</th>
<th>_id 30</th>
<th>_id 27</th>
</tr>
</thead>
<tbody>
<tr>
<td>rec 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rec 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rec 3</td>
<td></td>
<td></td>
<td>rec 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rec(n-1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rec (n)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure** – MongoDB scanning table Vs records’ order on the disk
Experimentation - MongoDB results

The exact issue
⇒ Mismatch between the scanning table vs data stored on disk

Collection _ids:
used by MongoDB process and WiredTiger

<table>
<thead>
<tr>
<th>_id 1</th>
<th>_id 2</th>
<th>_id 3</th>
<th>_id 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>_id 7</td>
<td>_id 8</td>
<td>_id 10</td>
<td>_id 11</td>
</tr>
<tr>
<td>_id 19</td>
<td>_id 20</td>
<td>_id 27</td>
<td>⋮</td>
</tr>
</tbody>
</table>

1) Get this doc

Collection file on disk:
allocated regarding the key sharding (hashed _id)

<table>
<thead>
<tr>
<th>_id 1</th>
<th>_id 3</th>
<th>_id 7</th>
<th>_id 5</th>
<th>_id 8</th>
<th>_id 10</th>
<th>_id 2</th>
<th>_id 19</th>
<th>_id 11</th>
<th>_id 20</th>
<th>_id 30</th>
<th>_id 27</th>
</tr>
</thead>
<tbody>
<tr>
<td>rec 1</td>
<td>rec 2</td>
<td>rec 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>rec(n-1)</td>
<td>rec(n)</td>
</tr>
</tbody>
</table>

**Figure** – MongoDB scanning table Vs records’ order on the disk
The exact issue

⇒ Mismatch between the scanning table vs data stored on disk

Collection _ids:
used by MongoDB process and WiredTiger

<table>
<thead>
<tr>
<th>_id 1</th>
<th>id 2</th>
<th>id 3</th>
<th>id 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>_id 7</td>
<td>id 8</td>
<td>id 10</td>
<td>id 11</td>
</tr>
<tr>
<td>_id 19</td>
<td>id 20</td>
<td>id 27</td>
<td>...</td>
</tr>
</tbody>
</table>

1) Get this doc
2) Which doc is next?

Collection file on disk:
allocated regarding the key sharding (hashed _id)

<table>
<thead>
<tr>
<th>_id 1</th>
<th>_id 3</th>
<th>_id 7</th>
<th>_id 5</th>
<th>_id 8</th>
<th>_id 10</th>
<th>_id 2</th>
<th>_id 19</th>
<th>_id 11</th>
<th>_id 20</th>
<th>_id 30</th>
<th>_id 27</th>
</tr>
</thead>
<tbody>
<tr>
<td>rec 1</td>
<td>rec 2</td>
<td>rec 3</td>
<td>_id 5</td>
<td>_id 8</td>
<td>_id 10</td>
<td>_id 2</td>
<td>_id 19</td>
<td>_id 11</td>
<td>_id 20</td>
<td>_id 30</td>
<td>_id 27</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure** – MongoDB scanning table Vs records’ order on the disk
The exact issue

⇒ Mismatch between the scanning table vs data stored on disk

1) Get this doc

Collection _ids:
used by MongoDB process and WiredTiger

<table>
<thead>
<tr>
<th>_id 1</th>
<th>id 2</th>
<th>id 3</th>
<th>id 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>_id 7</td>
<td>id 8</td>
<td>id 10</td>
<td>id 11</td>
</tr>
<tr>
<td>_id 19</td>
<td>id 20</td>
<td>id 27</td>
<td>...</td>
</tr>
</tbody>
</table>

Collection file on disk:
allocated regarding the key sharding (hashed _id)

<table>
<thead>
<tr>
<th>_id 1</th>
<th>_id 3</th>
<th>_id 7</th>
<th>_id 5</th>
<th>_id 8</th>
<th>_id 10</th>
<th>_id 2</th>
<th>_id 19</th>
<th>_id 11</th>
<th>_id 20</th>
<th>_id 30</th>
<th>_id 27</th>
</tr>
</thead>
<tbody>
<tr>
<td>rec 1</td>
<td>rec 2</td>
<td>rec 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>rec(n-1)</td>
<td>rec (n)</td>
</tr>
</tbody>
</table>

Figure – MongoDB scanning table Vs records’ order on the disk
The exact issue

⇒ Mismatch between the scanning table vs data stored on disk

Collection _ids:
used by MongoDB process and WiredTiger

<table>
<thead>
<tr>
<th>_id 1</th>
<th>_id 2</th>
<th>_id 3</th>
<th>_id 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>_id 7</td>
<td>_id 8</td>
<td>_id 10</td>
<td>_id 11</td>
</tr>
<tr>
<td>_id 19</td>
<td>_id 20</td>
<td>_id 27</td>
<td>...</td>
</tr>
</tbody>
</table>

1) Get this doc
2) Which doc is next?

Collection file on disk:
allocated regarding the key sharding (hashed _id)

<table>
<thead>
<tr>
<th>_id 1</th>
<th>_id 3</th>
<th>_id 7</th>
<th>_id 5</th>
<th>_id 8</th>
<th>_id 10</th>
<th>_id 2</th>
<th>_id 19</th>
<th>_id 11</th>
<th>_id 20</th>
<th>_id 30</th>
<th>_id 27</th>
</tr>
</thead>
<tbody>
<tr>
<td>rec 1</td>
<td>rec 2</td>
<td>rec 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>rec(n-1)</td>
<td>rec (n)</td>
</tr>
</tbody>
</table>

**Figure** — MongoDB scanning table Vs records’ order on the disk
The exact issue

⇒ Mismatch between the scanning table vs data stored on disk

Collection _ids:
used by MongoDB process and WiredTiger

<table>
<thead>
<tr>
<th>_id 1</th>
<th>_id 2</th>
<th>_id 3</th>
<th>_id 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>_id 7</td>
<td>_id 8</td>
<td>_id 10</td>
<td>_id 11</td>
</tr>
<tr>
<td>_id 19</td>
<td>_id 20</td>
<td>_id 27</td>
<td>...</td>
</tr>
</tbody>
</table>

1) Get this doc

Collection file on disk:
allocated regarding the key sharding (hashed _id)

<table>
<thead>
<tr>
<th>_id 1</th>
<th>_id 3</th>
<th>_id 7</th>
<th>_id 5</th>
<th>_id 8</th>
<th>_id 10</th>
<th>_id 2</th>
<th>_id 19</th>
<th>_id 11</th>
<th>_id 20</th>
<th>_id 30</th>
<th>_id 27</th>
</tr>
</thead>
<tbody>
<tr>
<td>rec 1</td>
<td>rec 2</td>
<td>rec 3</td>
<td>rec 4</td>
<td>rec 5</td>
<td>rec 6</td>
<td>rec 7</td>
<td>rec 8</td>
<td>rec 9</td>
<td>rec 10</td>
<td>rec 11</td>
<td>rec 12</td>
</tr>
</tbody>
</table>

Figure – MongoDB scanning table Vs records’ order on the disk
The exact issue
⇒ Mismatch between the scanning table vs data stored on disk

Collection _ids:
used by MongoDB process and WiredTiger

1) Get this doc
2) Which doc is next?

Collection file on disk:
allocated regarding the key sharding (hashed _id)

Figure – MongoDB scanning table Vs records’ order on the disk
The exact issue
⇒ Mismatch between the scanning table vs data stored on disk

Collection _ids:
used by MongoDB process and WiredTiger

1) Get this doc

Collection file on disk:
allocated regarding the key sharding (hashed _id)

Figure – MongoDB scanning table Vs records’ order on the disk
Experimentation - MongoDB results

**The exact issue**

$\Rightarrow$ Mismatch between the scanning table vs data stored on disk

Collection _ids:
used by MongoDB process and WiredTiger

<table>
<thead>
<tr>
<th>_id 1</th>
<th>_id 2</th>
<th>_id 3</th>
<th>_id 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>_id 7</td>
<td>_id 8</td>
<td>_id 10</td>
<td>_id 11</td>
</tr>
<tr>
<td>_id 19</td>
<td>_id 20</td>
<td>_id 27</td>
<td>...</td>
</tr>
</tbody>
</table>

1) Get this doc
2) Which doc is next?

Collection file on disk:
allocated regarding the key sharding (hashed _id)

<table>
<thead>
<tr>
<th>_id 1</th>
<th>_id 3</th>
<th>_id 7</th>
<th>_id 5</th>
<th>_id 8</th>
<th>_id 10</th>
<th>_id 2</th>
<th>_id 19</th>
<th>_id 11</th>
<th>_id 20</th>
<th>_id 30</th>
<th>_id 27</th>
</tr>
</thead>
<tbody>
<tr>
<td>rec 1</td>
<td>rec 2</td>
<td>rec 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure** - MongoDB scanning table Vs records’ order on the disk
The exact issue
⇒ Mismatch between the scanning table vs data stored on disk

![Diagram of MongoDB scanning table Vs records’ order on the disk]

**Figure** – MongoDB scanning table Vs records’ order on the disk
The exact issue

⇒ Mismatch between the scanning table vs data stored on disk

1) Get this doc
2) Which doc is next?

Collection _ids:
used by MongoDB process and WiredTiger

Collection file on disk:
allocated regarding the key sharding (hashed _id)

FIGURE – MongoDB scanning table Vs records’ order on the disk
The exact issue

⇒ Mismatch between the scanning table vs data stored on disk

Collection _ids:
used by MongoDB process and WiredTiger

1) Get this doc

Collection file on disk:
allocated regarding the key sharding (hashed _id)

**Figure** – MongoDB scanning table Vs records’ order on the disk
The exact issue
⇒ Mismatch between the scanning table vs data stored on disk

Collection _ids:
used by MongoDB process and WiredTiger

1) Get this doc
2) Which doc is next?

Collection file on disk:
allocated regarding the key sharding (hashed _id)

Figure – MongoDB scanning table Vs records’ order on the disk
The exact issue

⇒ Mismatch between the scanning table vs data stored on disk

Collection _ids:
used by MongoDB process and WiredTiger

Collection file on disk:
allocated regarding the key sharding (hashed _id)

1) Get this doc

**Figure** – MongoDB scanning table Vs records’ order on the disk
The exact issue
⇒ Mismatch between the scanning table vs data stored on disk

Figure – MongoDB scanning table Vs records’ order on the disk
The exact issue
⇒ Mismatch between the scanning table vs data stored on disk

Collection _ids:
used by MongoDB process and WiredTiger

Collection file on disk:
allocated regarding the key sharding (hashed _id)

1) Get this doc
2) Which doc is next?

Figure – MongoDB scanning table Vs records’ order on the disk
The exact issue

⇒ Mismatch between the scanning table vs data stored on disk

Collection _ids:
used by MongoDB process and WiredTiger

<table>
<thead>
<tr>
<th>_id 1</th>
<th>_id 2</th>
<th>_id 3</th>
<th>_id 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>_id 7</td>
<td>_id 8</td>
<td>_id 10</td>
<td>_id 11</td>
</tr>
<tr>
<td>_id 19</td>
<td>_id 20</td>
<td>_id 27</td>
<td>...</td>
</tr>
</tbody>
</table>

1) Get this doc
2) Which doc is next?

Collection file on disk:
allocated regarding the key sharding (hashed _id)

<table>
<thead>
<tr>
<th>_id 1</th>
<th>_id 3</th>
<th>_id 7</th>
<th>_id 5</th>
<th>_id 8</th>
<th>_id 10</th>
<th>_id 2</th>
<th>_id 19</th>
<th>_id 11</th>
<th>_id 20</th>
<th>_id 30</th>
<th>_id 27</th>
</tr>
</thead>
<tbody>
<tr>
<td>rec 1</td>
<td>rec 2</td>
<td>rec 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>rec(n-1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>rec(n)</td>
<td></td>
</tr>
</tbody>
</table>

Figure – MongoDB scanning table Vs records’ order on the disk

We proposed an ad-hoc solution

⇒ Key-idea: rewrite shards data
⇒ MongoDB updates its view of data
⇒ High cost, but gives insights!
Conclusions

⇒ Low-level I/O evaluations are not negligible to preserve I/O performance
⇒ Systems’ complexity may hide issues
  → We showed how an unexpected issue affects the performance of MongoDB
⇒ IOscope is proposed to analyse I/O patterns of storage systems
⇒ We demonstrated how it is worthy to use IOscope to go beyond high-level evaluations’ results

Future work

⇒ Extend IOscope to uncover other I/O-related issues
⇒ Performing more performance evaluations on other storage systems
⇒ Further investigation on SSDs and I/O patterns
Conclusions

⇒ Low-level I/O evaluations are not negligible to preserve I/O performance
⇒ Systems’ complexity may hide issues
  → We showed how an unexpected issue affects the performance of MongoDB
⇒ IOscope is proposed to analyse I/O patterns of storage systems
⇒ We demonstrated how it is worthy to use IOscope to go beyond high-level evaluations’ results

Future work

⇒ Extend IOscope to uncover other I/O-related issues
⇒ Performing more performance evaluations on other storage systems
⇒ Further investigation on SSDs and I/O patterns

Questions are welcome!