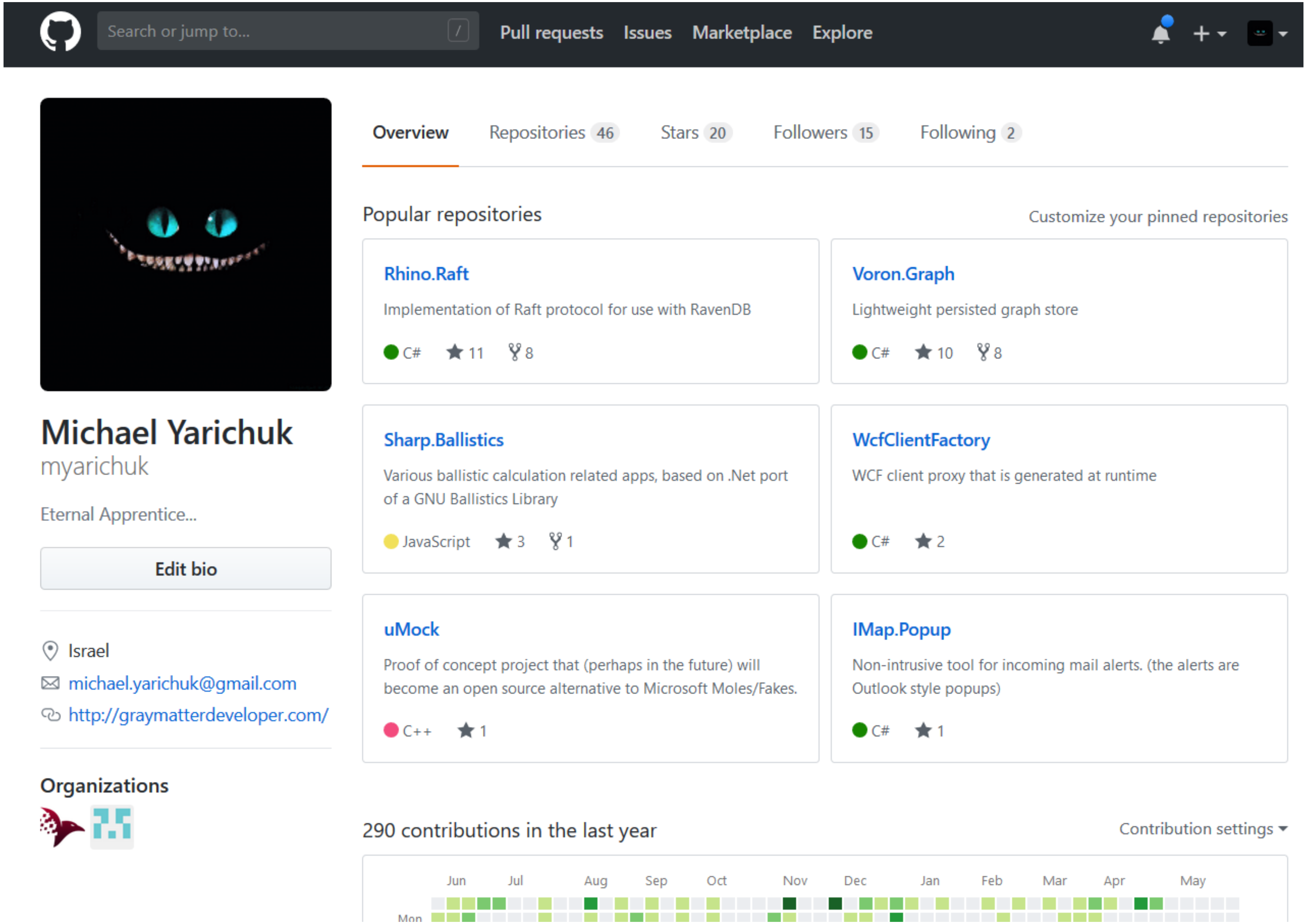
Why databases cry at night

Michael Yarichuk

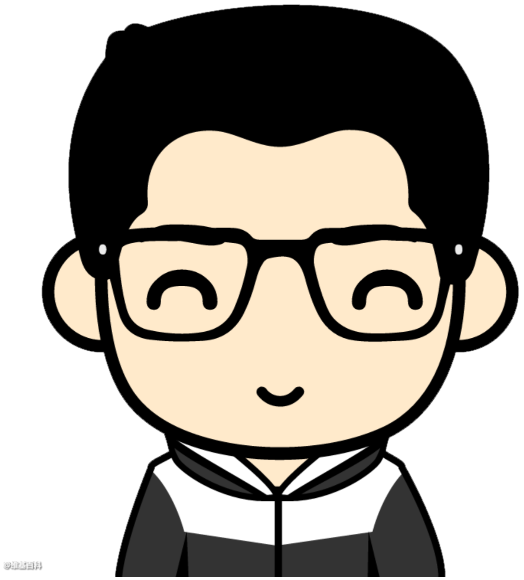


Search or jump to...
Pull requests
Issues
Marketplace
Explore
Overview
Repositories 46
Stars 20
Followers 15
Following 2
Michael Yarichuk
myarichuk
Eternal Apprentice...
Edit bio
Israel
michael.yarichuk@gmail.com
http://graymatterdeveloper.com/
Organizations
Popular repositories
Customize your pinned repositories
Rhino.Raft
Implementation of Raft protocol for use with RavenDB
C# 11 8
Voron.Graph
Lightweight persisted graph store
C# 10 8
Sharp.Ballistics
Various ballistic calculation related apps, based on .Net port of a GNU Ballistics Library
JavaScript 3 1
WcfClientFactory
WCF client proxy that is generated at runtime
C# 2
uMock
Proof of concept project that (perhaps in the future) will become an open source alternative to Microsoft Moles/Fakes.
C++ 1
IMap.Popup
Non-intrusive tool for incoming mail alerts. (the alerts are Outlook style popups)
C# 1
290 contributions in the last year
Contribution settings
Jun Jul Aug Sep Oct Nov Dec Jan Feb Mar Apr May
Mon

RAVENDB

Safe by Default, Optimized for Efficiency

Magic?



Nope. Not magic!

It’s only an abstraction…

The Law of Leaky Abstractions

"All non-trivial abstractions, to some degree, are leaky."

- Joel Spolsky

The issue types

- ❑Storage & relevant algorithms
- ❑Indexing & Queries
- ❑Network

1-a) Storage

It just works, no?

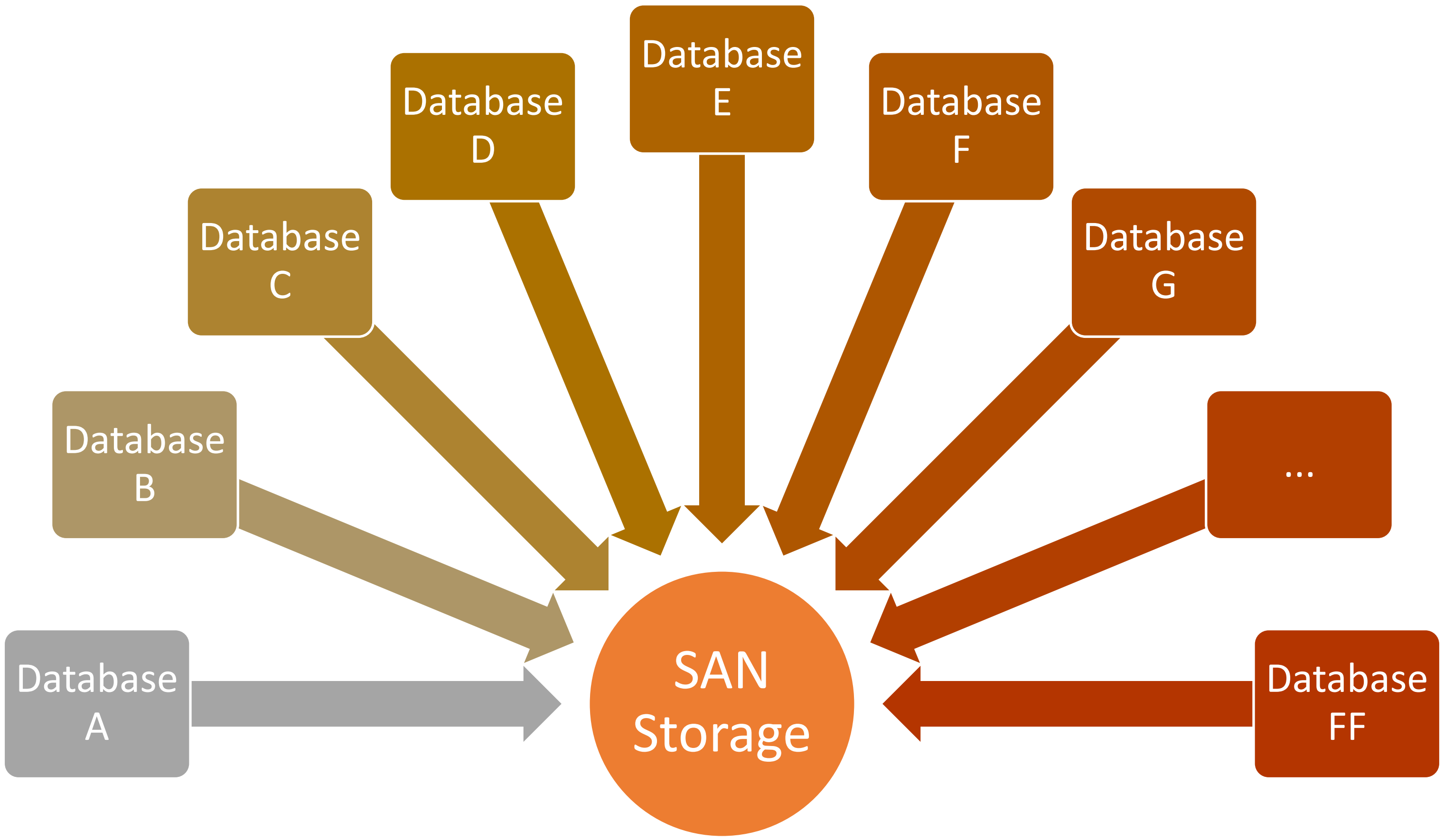
Here is a riddle… hint: storage!

RavenDB server-wide backup failed

- The instance had multiple databases in single instance
- Plenty of memory and cores, resource usage is small
- Nothing else was running on the machine EXCEPT RavenDB
- Scheduled backup tasks fail soon after they started

The backup tasks *started* at the *same time*!

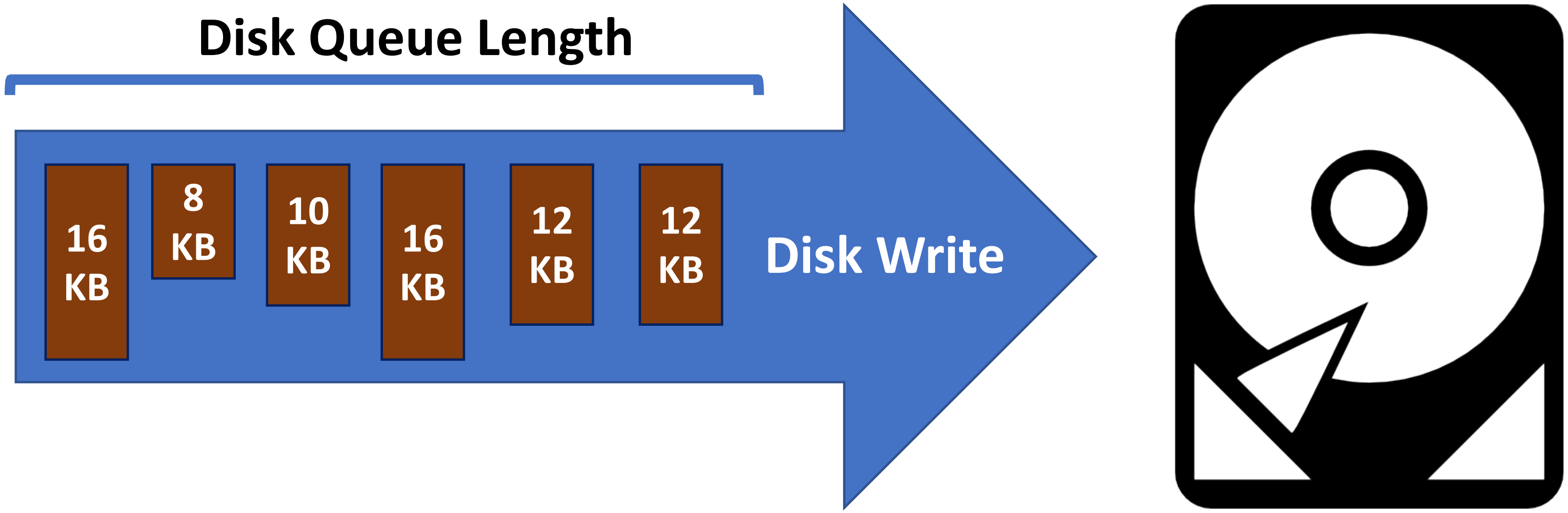
Also, there were gazillion of databases



Investigation: disk queue length

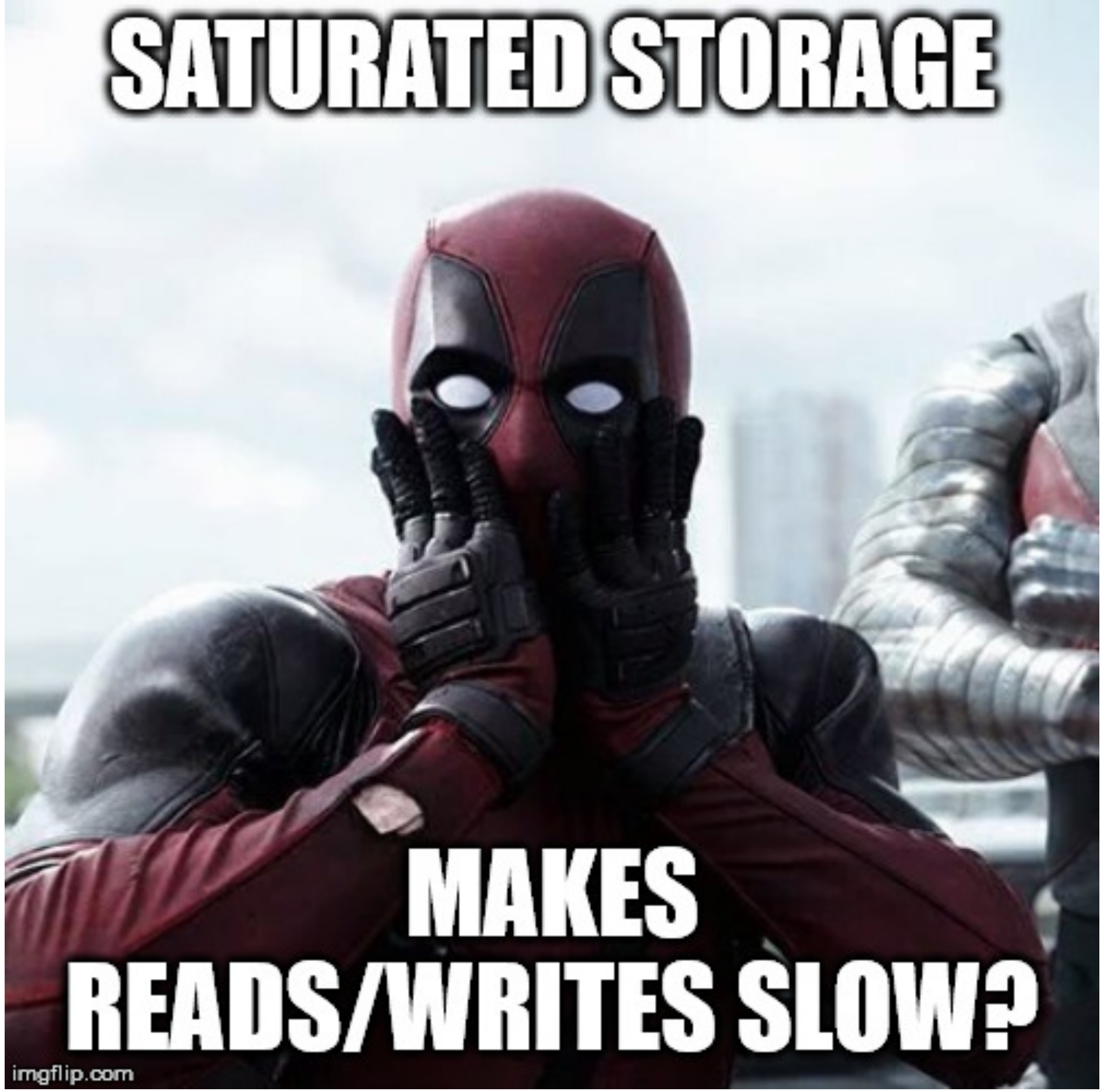
Bottleneck indicator

Active Time (%)	Available Space...	Total Space (MB)	Disk Queue Length
100.00	57,338	237,860	32.06
0.12	-	-	0.00



Disk Queue Length – what and how?

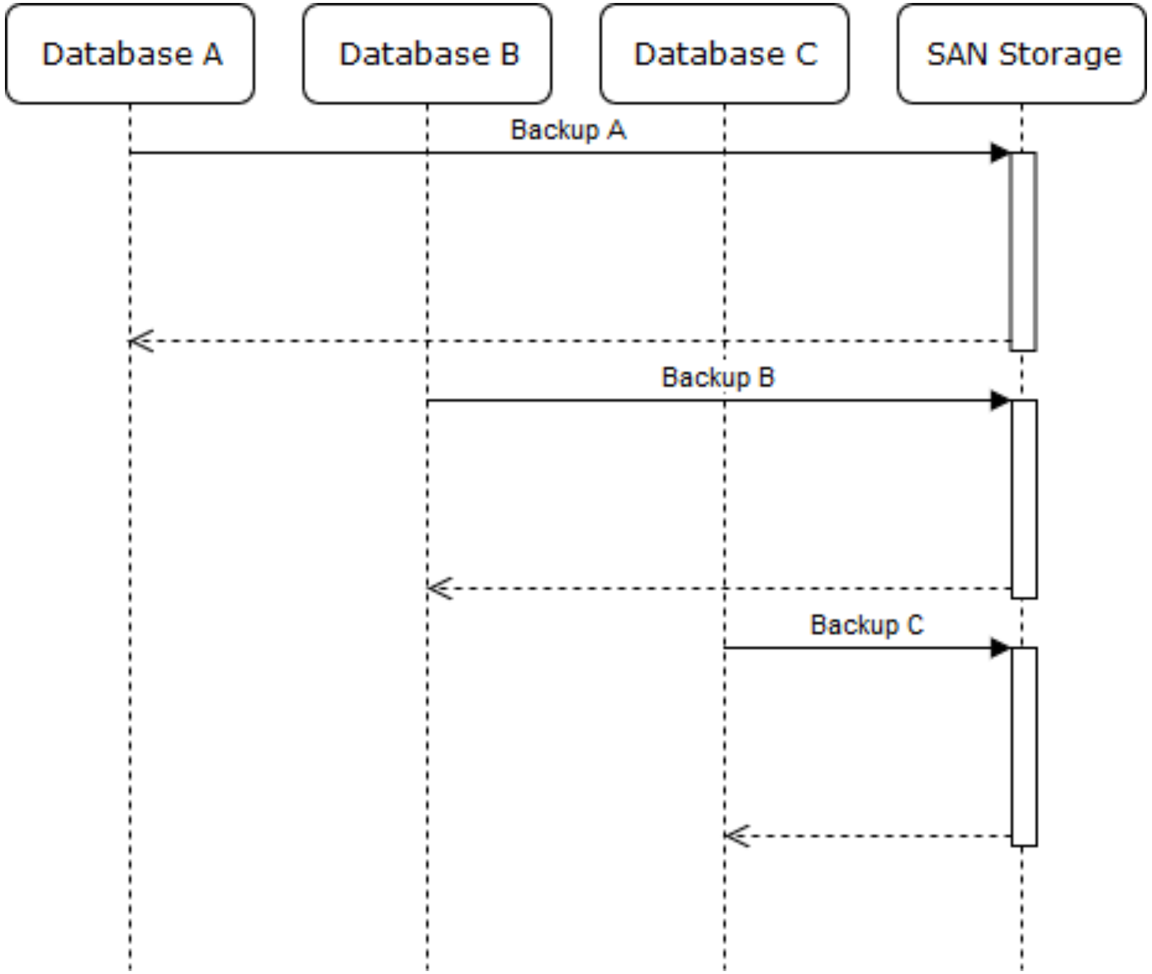
- Dashboards
- Monitoring
- Windows Perf Monitor


SATURATED STORAGE
MAKES
READS/WRITES SLOW?
imgflip.com

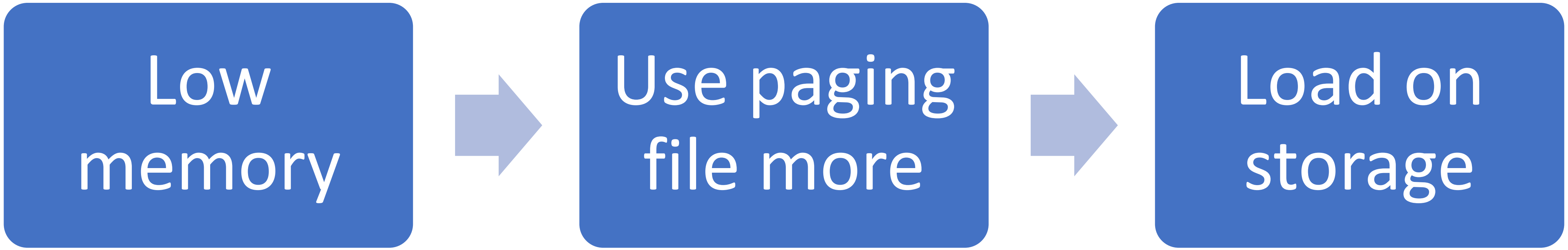
RavenDB's failing backups

Approx. 200 databases doing backups at the same time WILL cause storage saturation!

The solution was rather simple



Also: low memory matters!



What can we do about storge issues?

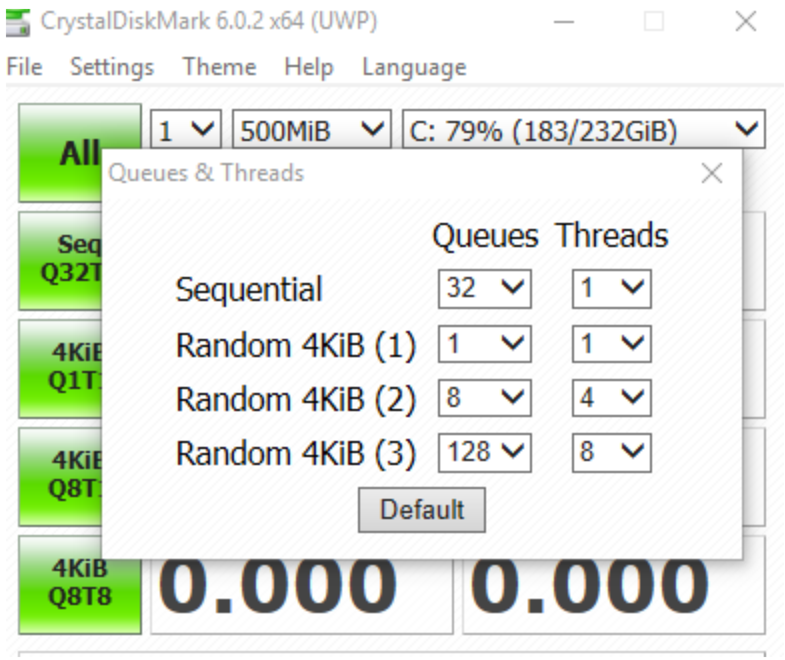
- Load test database-related code
  - **Write-through** throughput
  - Enough IOPS for expected production load (disk queue length is <= 2)
- Cloud → provision IOPS (ensure disk performance)
- Load-test application to find limits of the system
- Make sure no low-memory situations happen

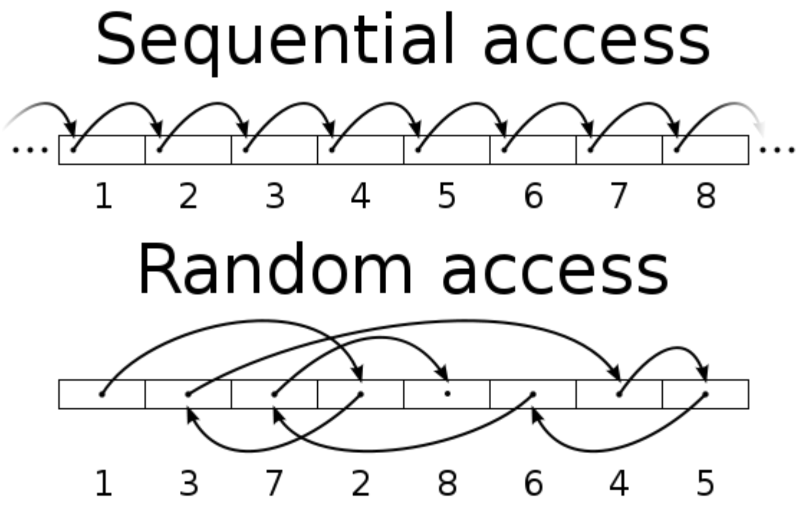
About storage benchmarks

- Sysinternals Process Monitor
- CrystalDiskMark
- ATTO Disk Benchmark
- (Many) other tools

CrystalDiskMark

- Random/sequential I/O?
- Queues/Threads (queue depth/length)
- Size of each read/write





We will discuss some more benchmarks later

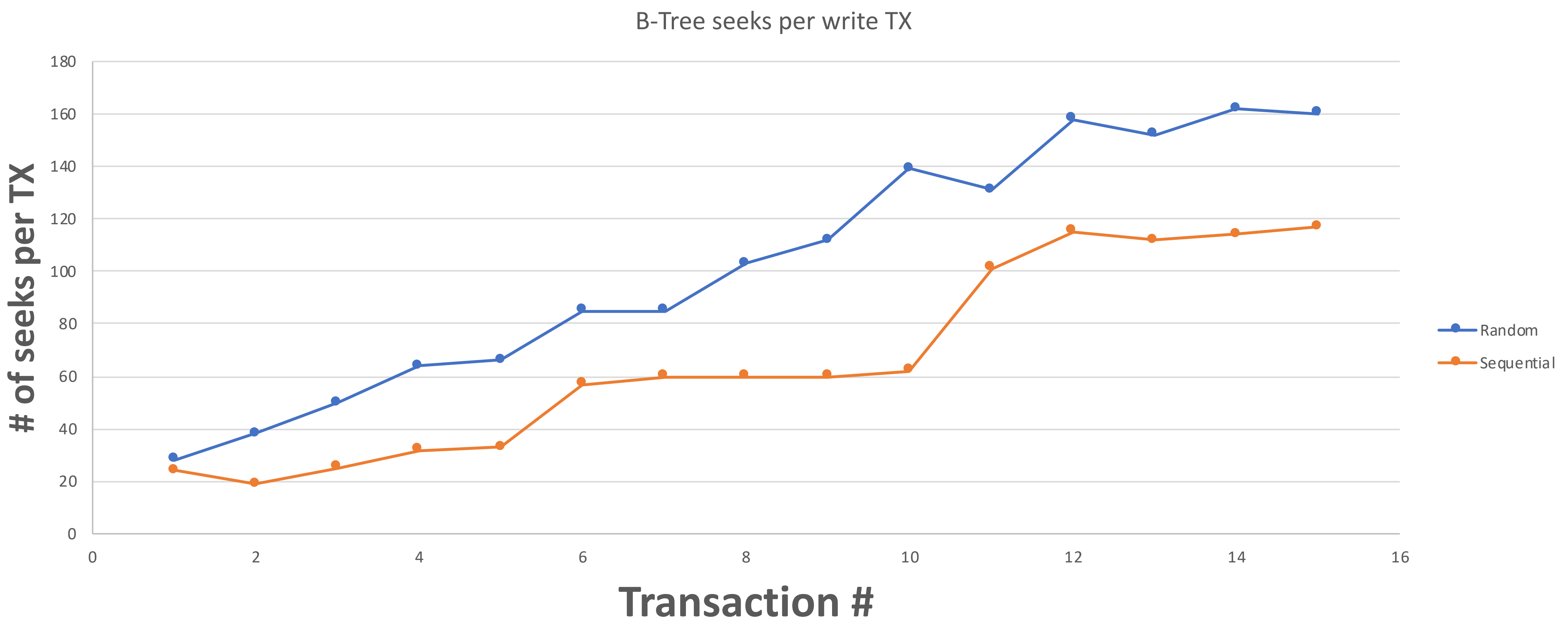
1-b) Storage

Effect of hardware of algorithm performance

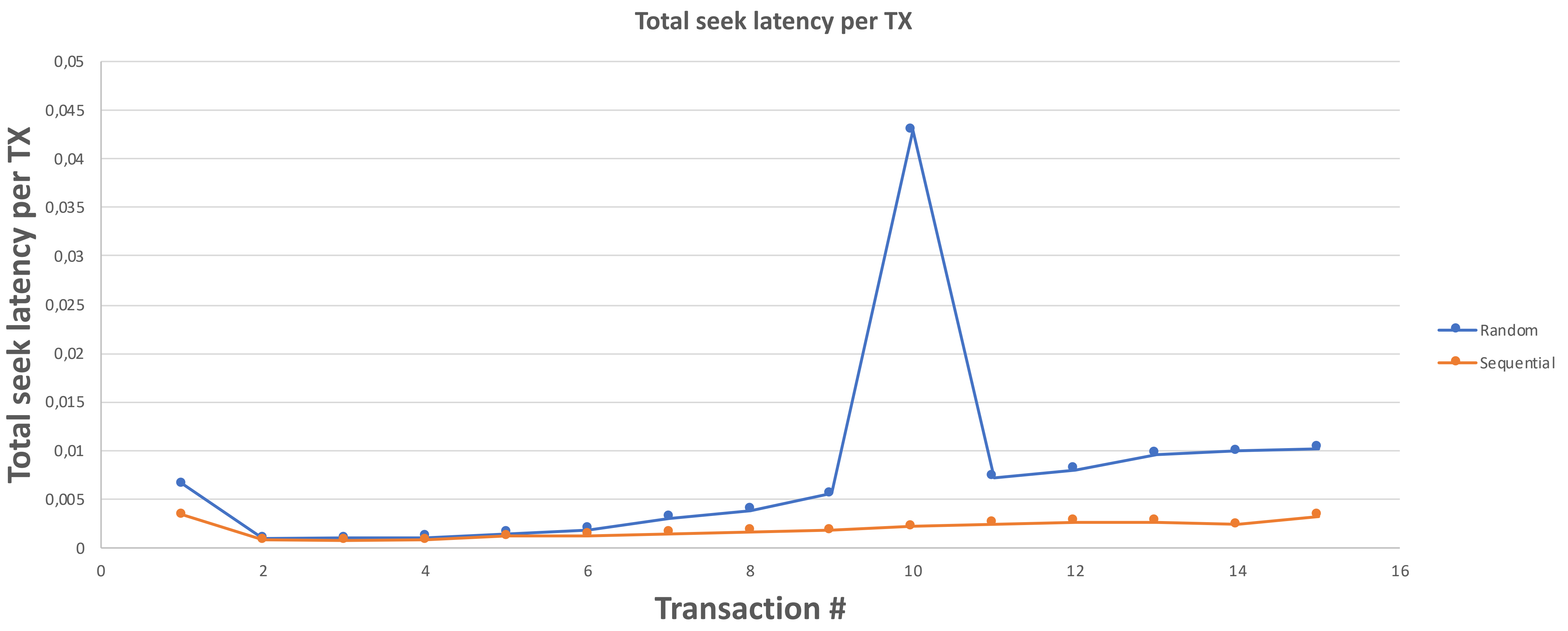
A tale of two primary keys

- One embedded transactional database engine (LMDB)
- 100 transactions, 100 key/value writes per transaction
- Two databases, keys and values have the same size
  - One uses sequential keys (using Win32’s ***UuidCreateSequential()***)
  - One uses random keys (using Win32 ***UuidCreate()***)

A tale of two primary keys



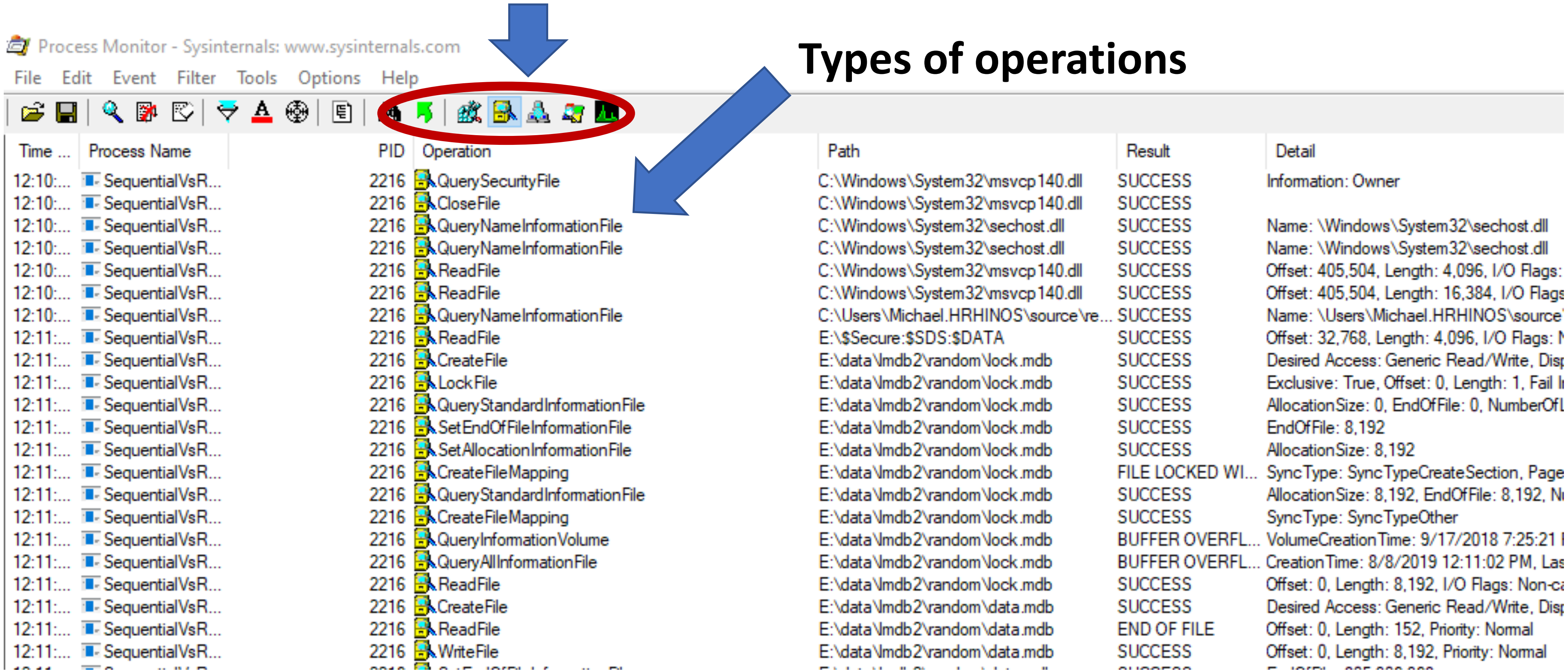
A tale of two primary keys



Process Monitor

Types of OS operations to listen (file activity, network activity, etc)

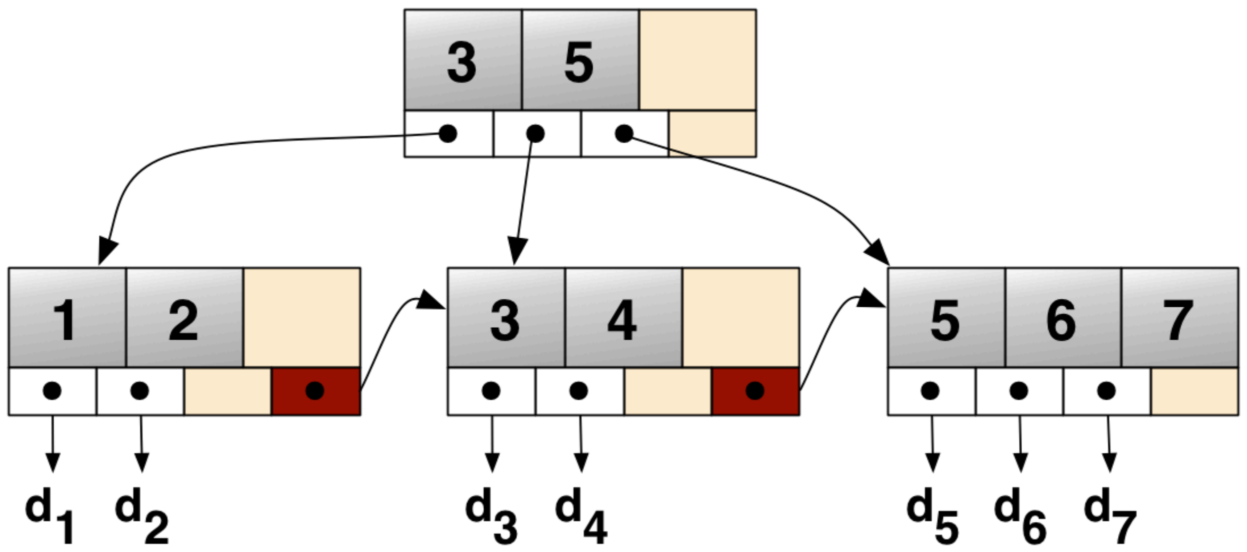
Types of operations



Why?

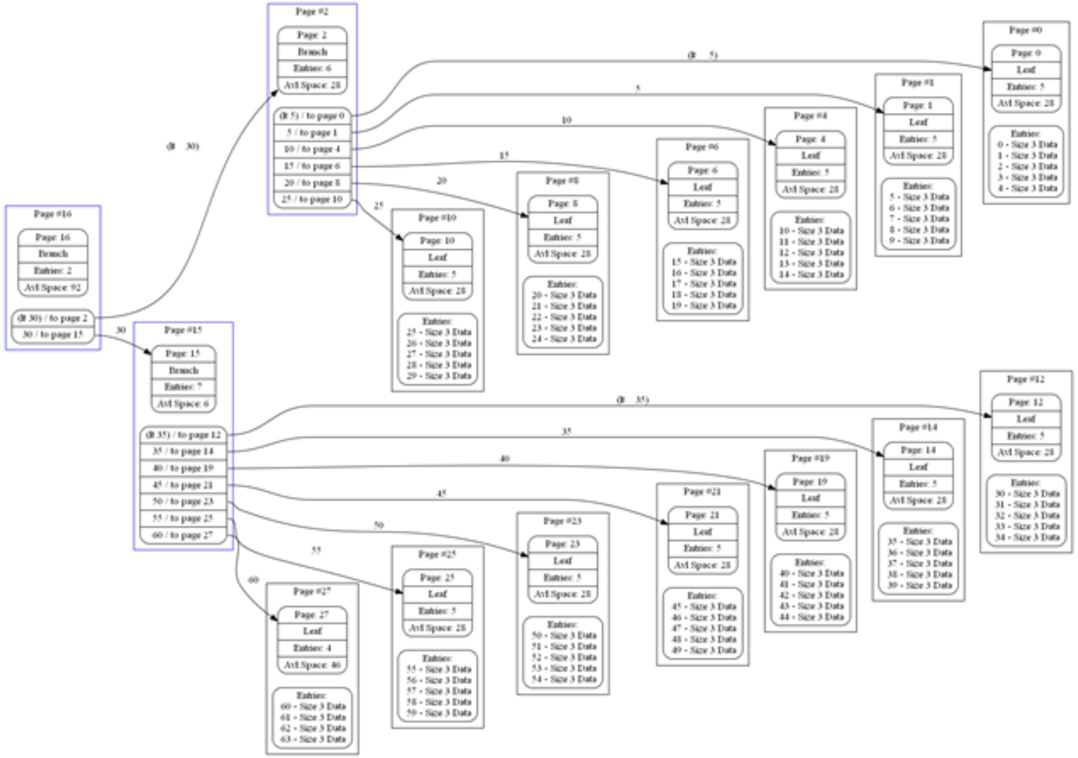
Storage algorithms (page-oriented storage engines)

- B-tree, B+ Tree
- Optimized for reads
- Optimized for sequential data

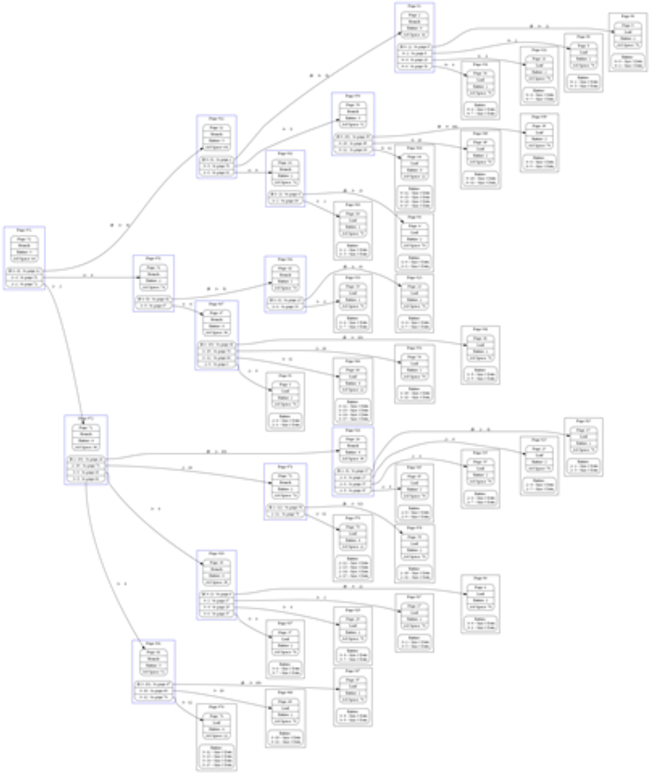


B+ tree keys

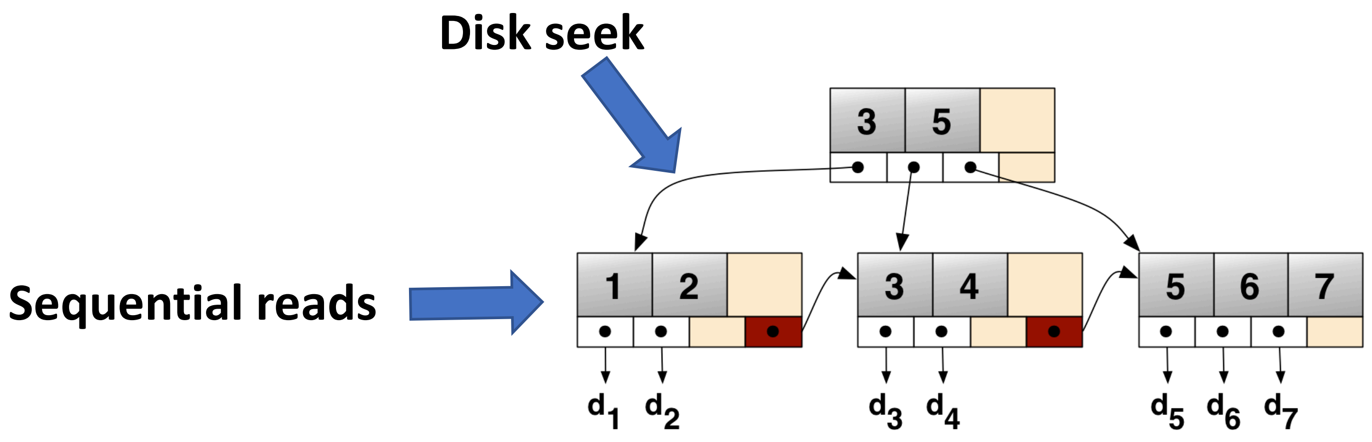
Sequential keys



Non-sequential keys



The cost of hops in the tree



SSD random read: 16,000ns ≈ 16µs

Disk seek: 3,000,000ns ≈ 3ms

Read 1,000,000 bytes sequentially from memory: 4,000ns ≈ 4µs

Read 1,000,000 bytes sequentially from disk: 947,000ns ≈ 947µs

https://people.eecs.berkeley.edu/~rcs/research/interactive_latency.html

Minimize performance impact of keys

- Sequential keys allow better performance
  - 1,2,3,4,5
  - Users/1, Users/2, Users/3
- B-Trees are used to store data AND indexes
  - Query performance!

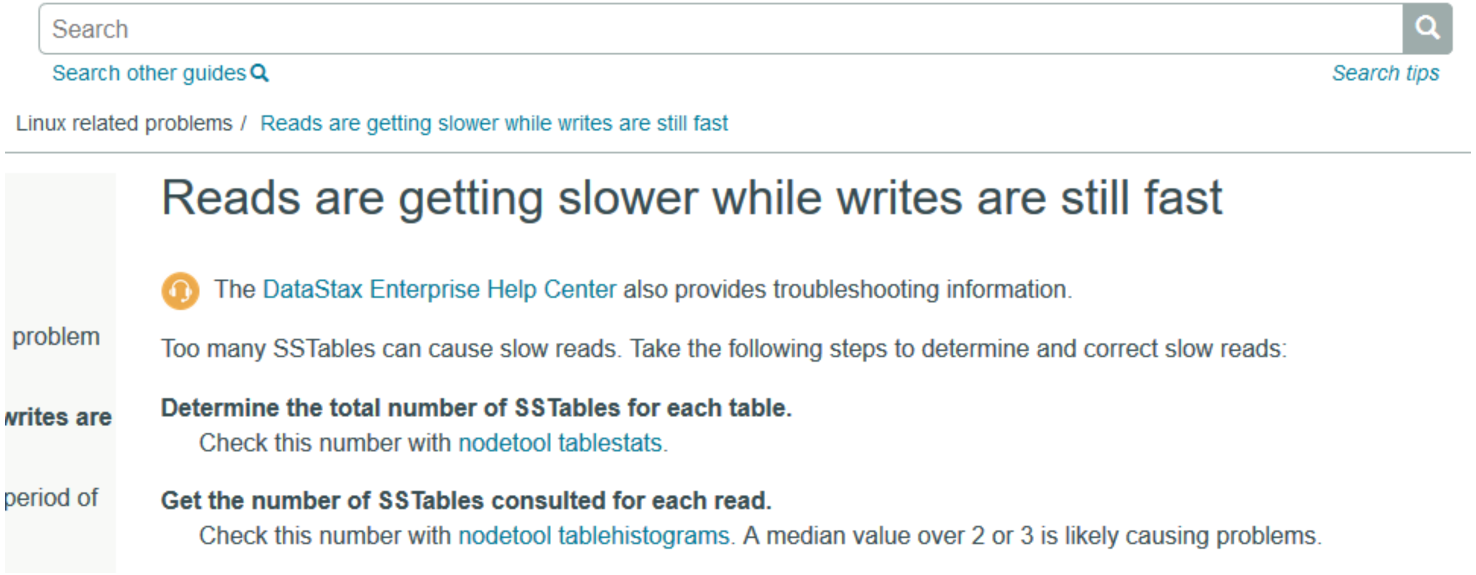
1-c) Storage

Data structure performance

The tale of an occasionally slow database

- Sometimes, Cassandra database was fast and sometimes not
- This happened non-deterministically

A page in Cassandra documentation!


Search
Search other guides
Search tips

Linux related problems / Reads are getting slower while writes are still fast

problem
writes are
period of

Reads are getting slower while writes are still fast

The DataStax Enterprise Help Center also provides troubleshooting information.

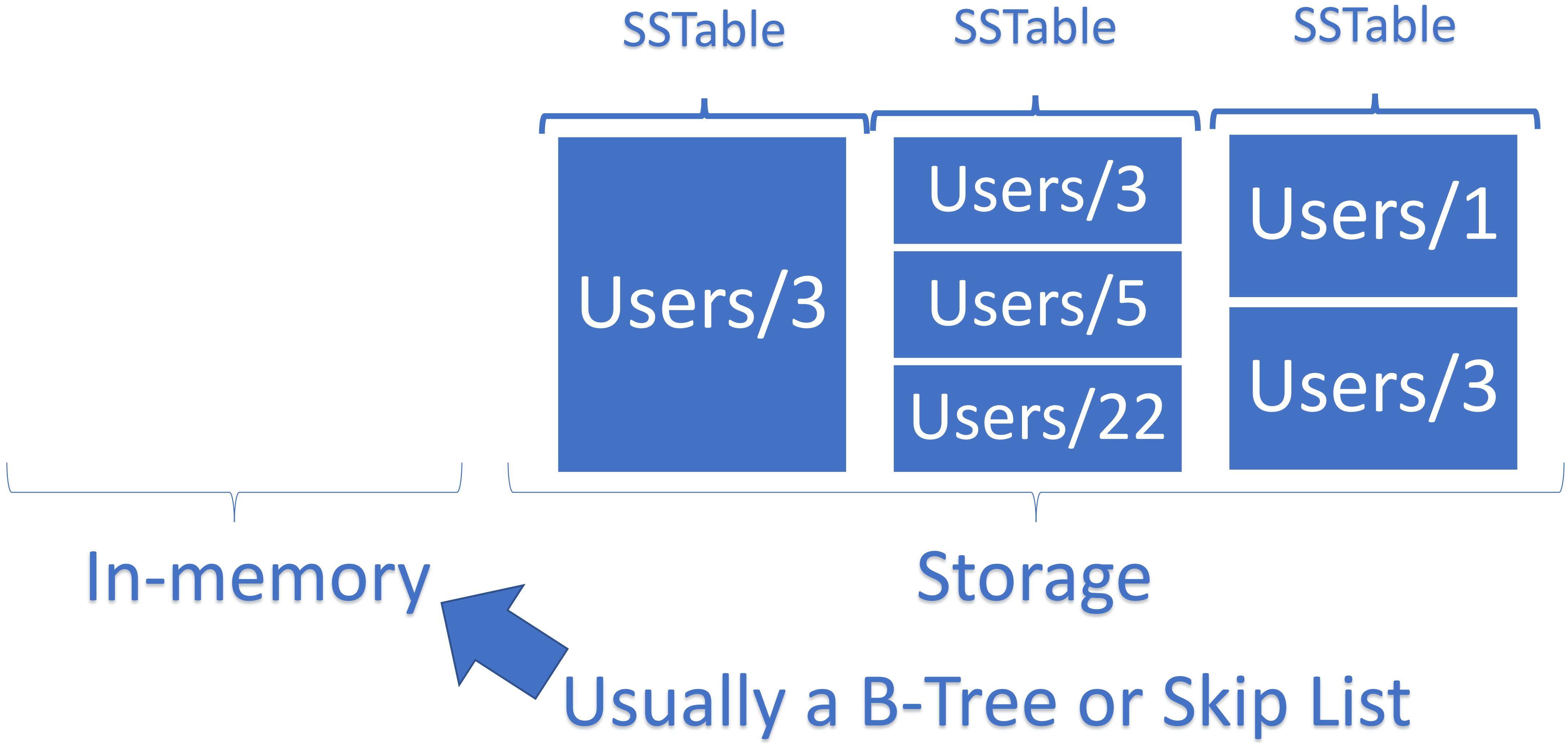
Too many SSTables can cause slow reads. Take the following steps to determine and correct slow reads:

Determine the total number of SSTables for each table.
Check this number with nodetool tablestats.

Get the number of SSTables consulted for each read.
Check this number with nodetool tablehistograms. A median value over 2 or 3 is likely causing problems.

https://docs.datastax.com/en/dse-trblshoot/doc/troubleshooting/slowReads.html

Storage algorithms (log-structured storage engines)



Storage algorithms (log-structured storage engines)

Insert ***users/44***

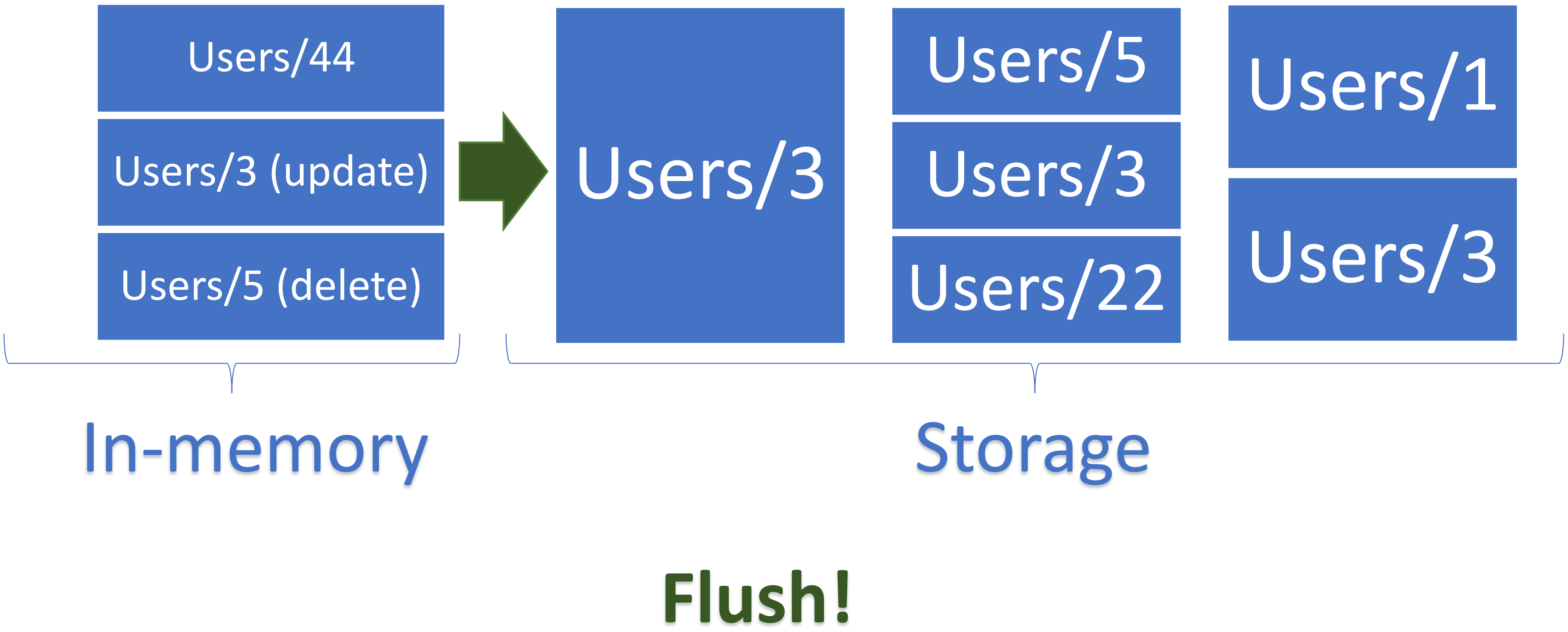
Storage algorithms (log-structured storage engines)

Update ***users/3***

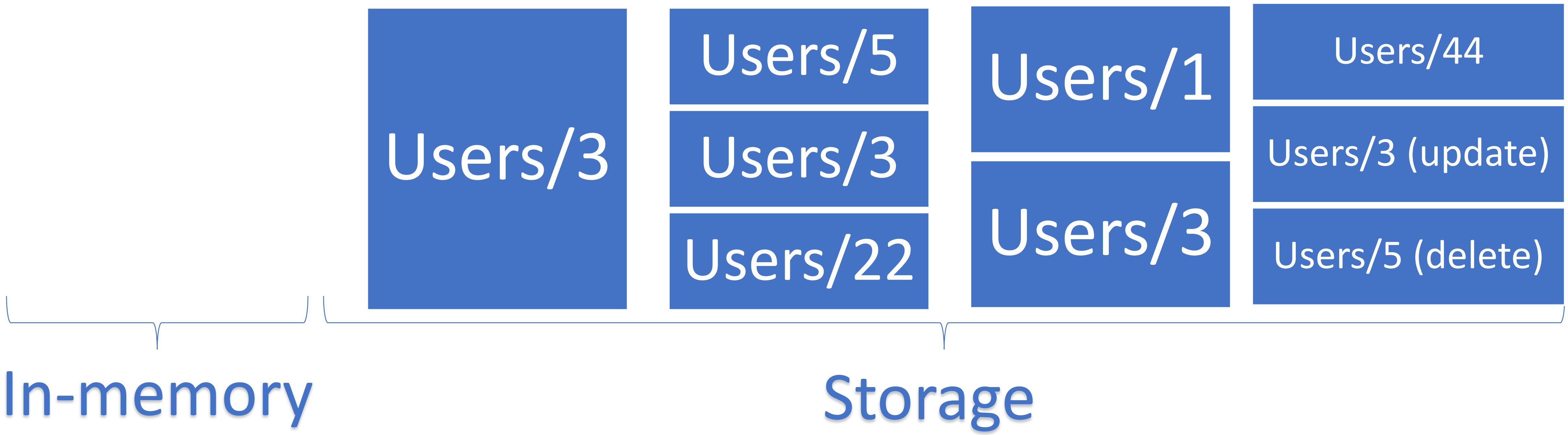
Storage algorithms (log-structured storage engines)

Delete ***users/5***

Storage algorithms (log-structured storage engines)

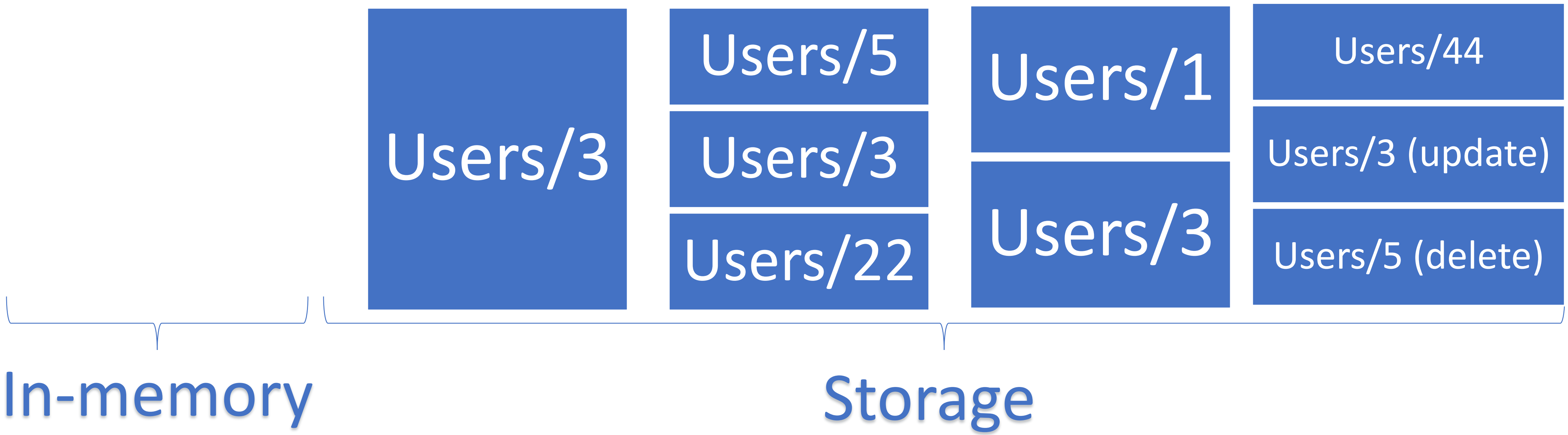


Storage algorithms (log-structured storage engines)



(appended at the end)

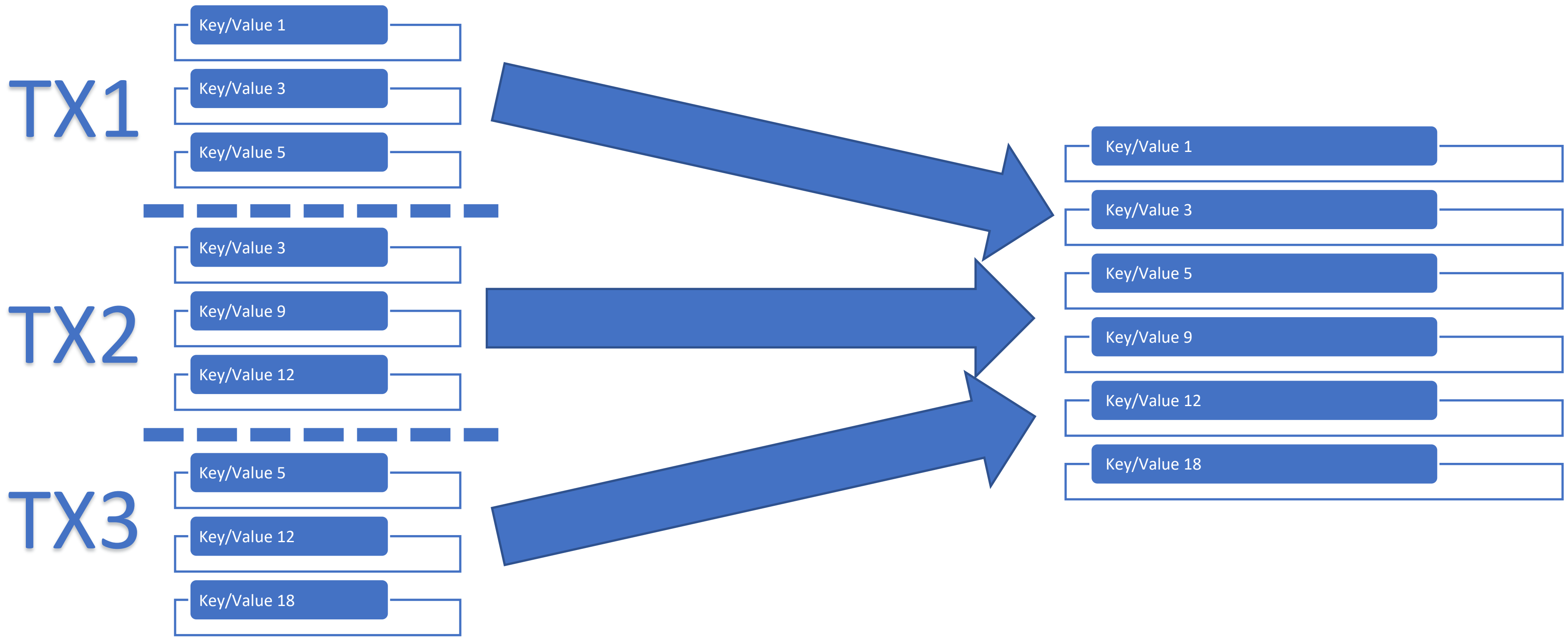
Storage algorithms (log-structured storage engines)



Reading requires searching ALL SSTables!

Storage algorithms – LSM Tree compaction

This is roughly O(n*log(n)) operation!



Storage algorithms – LSM Tree compaction

After compaction

Perfor… ce…

- Compa…
  - Time-…

Optimization for different usage patterns!

```
ALTER TABLE users
WITH compaction =
{'class' : 'SizeTieredCompactionStrategy', 'min_threshold' : 6 }
```

Different databases have specific options to optimize performance

For example

- MS-SQL index optimization
- RavenDB custom indexes
- MongoDB Aggregation Pipelines

1-d) Storage

Transaction implementation & performance

ACID guarantees!

All operations either succeed or not

- Atomicity

Data is consistent BEFORE and AFTER transaction

- Consistency

Multiple transactions do not interfere each other

- Isolation

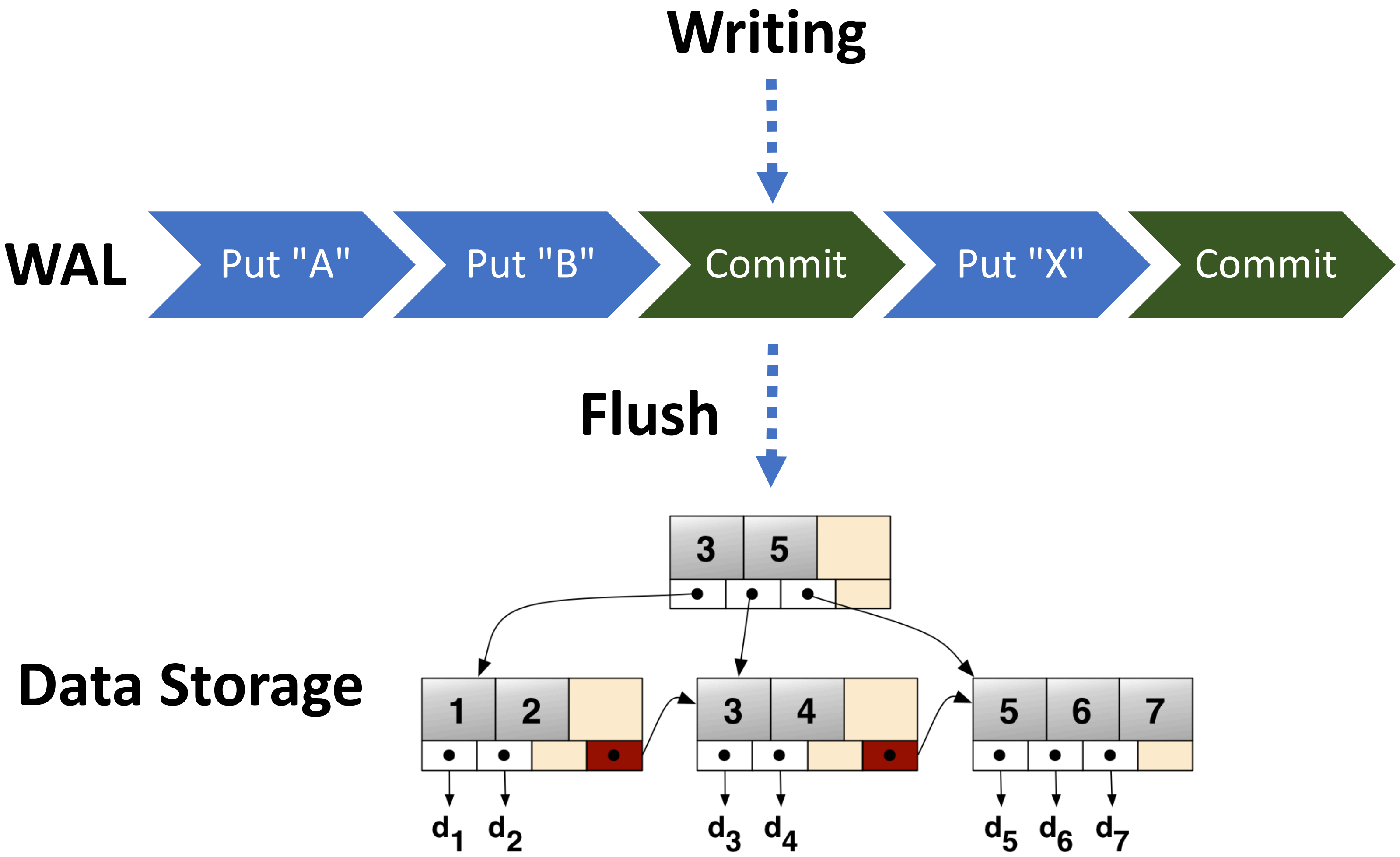
Even if system failure happens, transaction is recorded

- Durability

ACID guarantees!

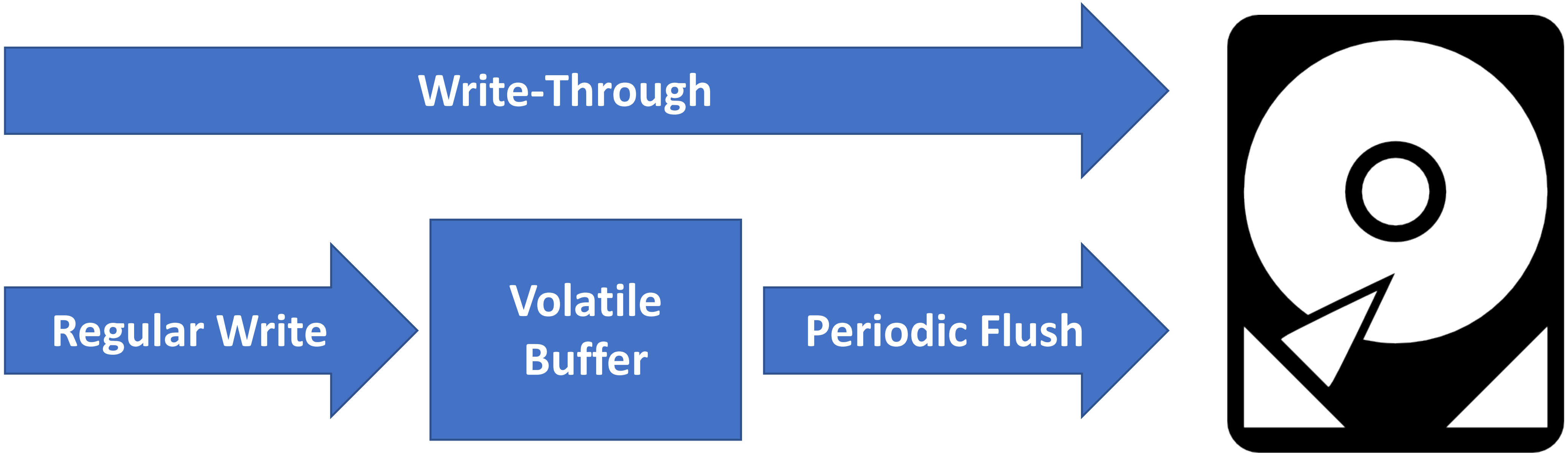
- Note: not all DBs support it
  - All of RDBMS
  - Some NoSQLs – RavenDB, LevelDB, LMDB

Write-ahead log (WAL) - Atomicity, Durability



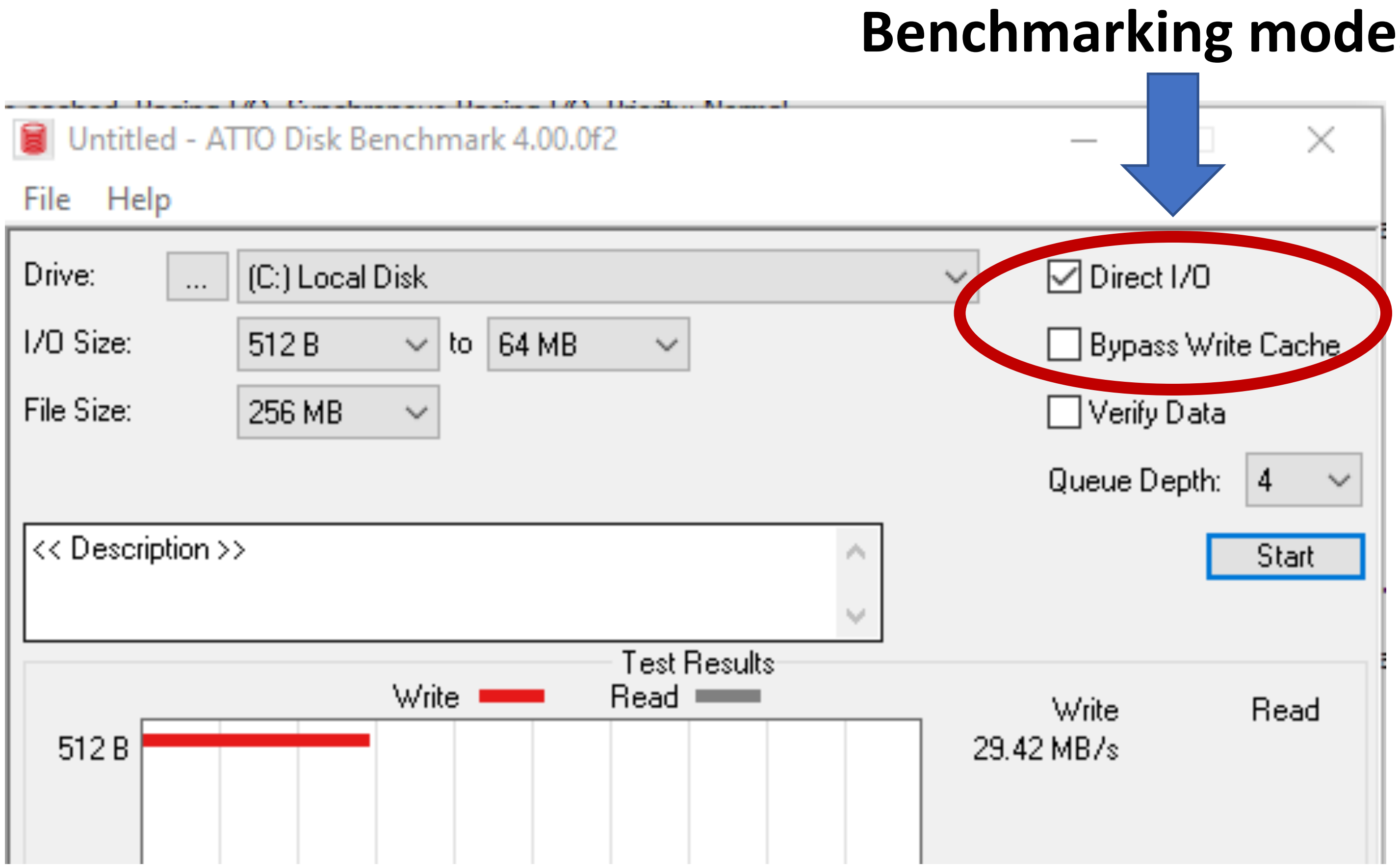
Write-ahead log (WAL)

- "Write-Through" writes – no caching (otherwise no durability!)
- Lots of small writes (overhead of each write)



Relevant storage benchmarks

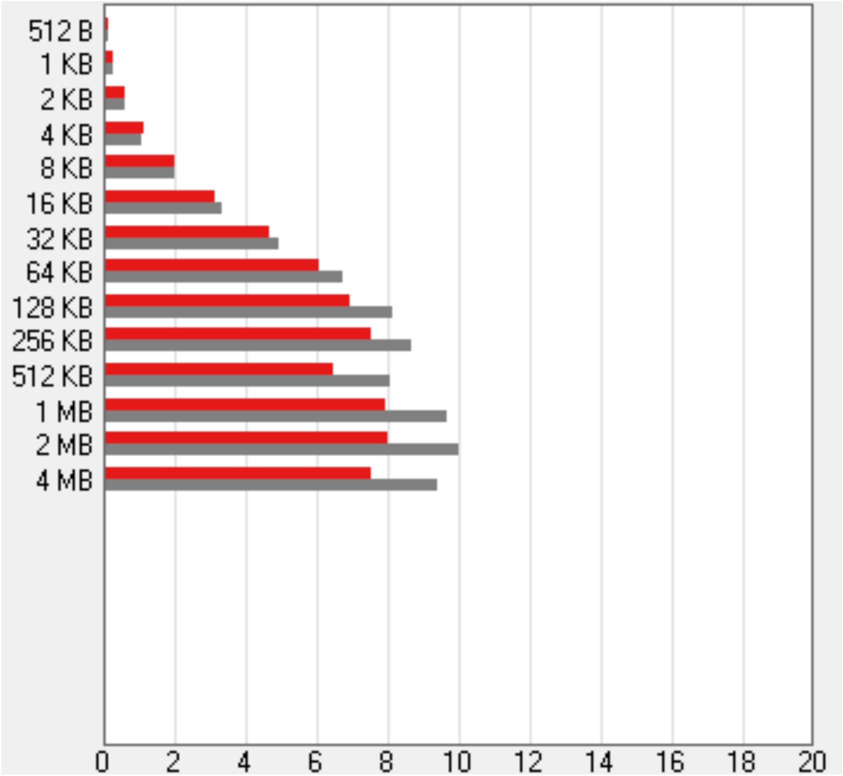
ATTO Disk Benchmark



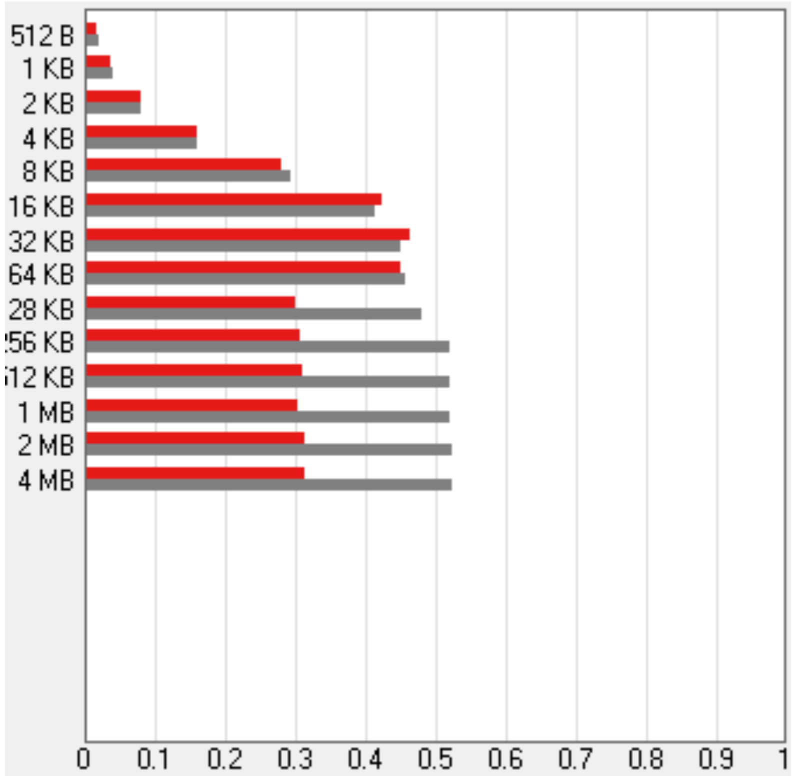
Buffered vs. Write-through

NVMe SSD (Samsung 860 EVO m.2)

With buffers and caching (GBs/sec)



No caching, Write-Through (MBs/sec)



Storage effect on transactions

- Slow storage throughput = bottleneck on transactions
- Write-through performance = transaction throughput

2-a) Indexing & Queries

Query complexity

Are those queries different?

Microsoft SQL Server

```sql
SELECT * FROM Orders
WHERE ShipToCity IN ("Paris", "Lyon") AND
      OrderedAt >= '2010-05-01'
```

mongoDB

```
db.Orders.find({
      "ShipTo.City": {
            "$in": ["Paris", "Lyon"]
      },
      "OrderedAt": {
            "$gte": "2010-05-1"
      }
});
```

RAVENDB
Safe by Default, Optimized for Efficiency

```
from Orders
where ShipTo.City in ("Paris", "Lyon") and
      OrderedAt >= '2010-05-01'
```

Let's start from something... simple.


```
{
    "Company": "companies/73-A",
    "Employee": "employees/7-A",
    "Freight": 18.44,
    "Lines": [
        {
            "Discount": 0.05,
            "PricePerUnit": 17.45,
            "Product": "products/16-A",
            "ProductName": "Pavlova",
            "Quantity": 14
        }
    ],
    "OrderedAt": "1998-05-06T00:00:00.0000000",
    "RequireAt": "1998-06-03T00:00:00.0000000",
    "ShipTo": {
        "City": "Kobenhavn",
        "Country": "Denmark",
        "Line1": "Vinbæltet 34",
        "Line2": null,
        "Location": null,
        "PostalCode": "1734",
        "Region": null
    },
```

First, we define an index

We create an index that covers **city** and **country** fields of *ShipTo*

```
> db.orders.getIndexes()
[
        {
                "v" : 2,
                "key" : {
                        "_id" : 1
                },
                "name" : "_id_",
                "ns" : "Northwind.orders"
        },
        {
                "v" : 2,
                "key" : {
                        "ShipTo.City" : 1,
                        "ShipTo.Country" : 1
                },
                "name" : "ShipToIndex",
                "ns" : "Northwind.orders"
        }
]
```

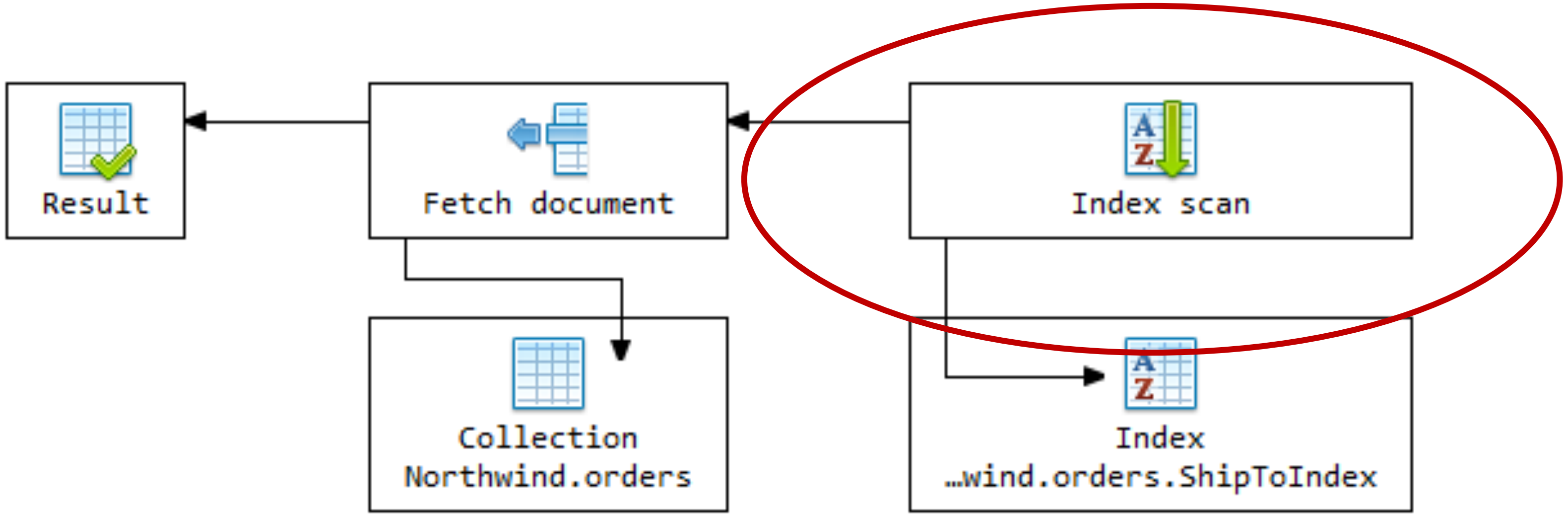
Then we do some queries

Fetching orders that were shipped to Paris or Lyon

```
db.orders.find({
    "ShipTo.City": {
        "$in": ["Paris", "Lyon"]
    }
});
```

mongoDB®

So far so good...



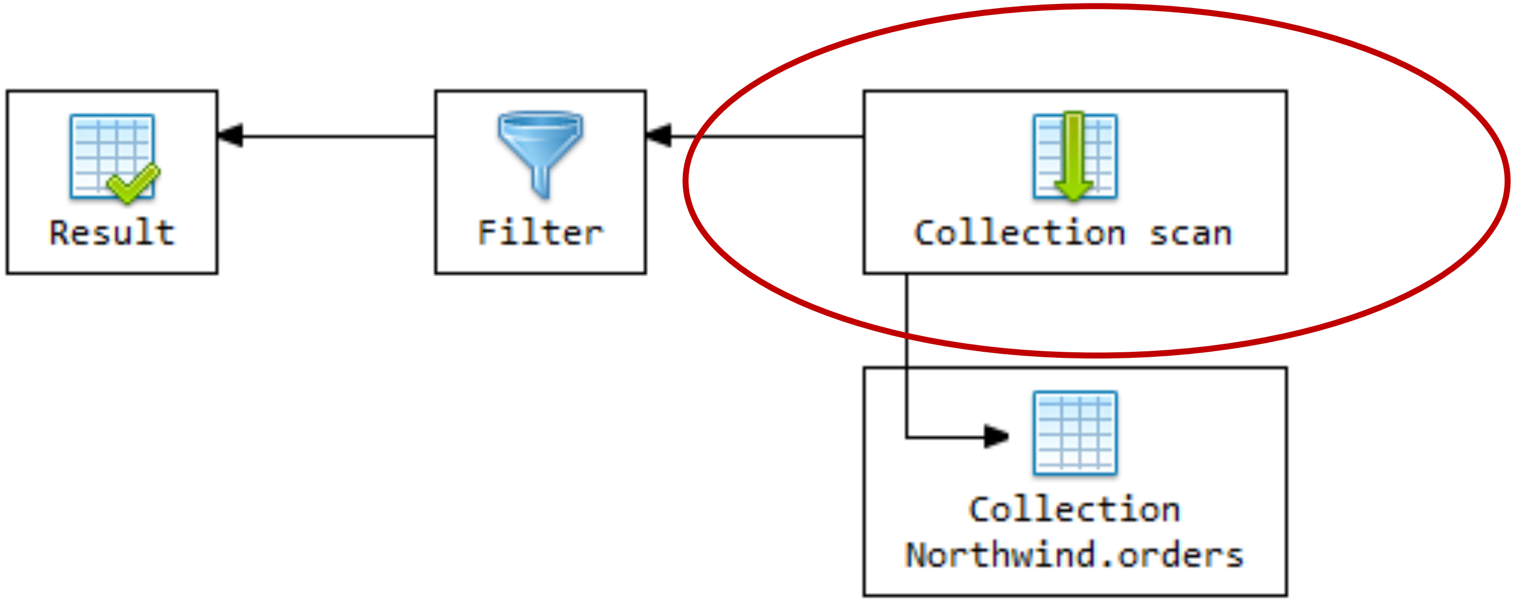
mongoDB

And another query

Fetching orders that were shipped to all of France

```
db.orders.find({
    "ShipTo.Country": "France"
});
```

It’s a gotcha!



Result
Filter
Collection scan
Collection
Northwind.orders

Index Scan vs. Collection Scan

Collection Scan

- O(n) scan

Index Scan

- O(log(n)) seek

Why?

In some databases (like MongoDB)

- Indexed fields are concatenated into single index key
- Filtering only by prefix

The values are concatenated!

Field 1	Field 2
Lyon	France
Paris	France
Oslo	Norway

Index Key	Record IDs
LyonFrance	[7],[1],[4]
ParisFrance	[5],[6]
OsloNorway	[12],[2],[9],[34]

Why?

In some databases (RavenDB, any Lucene-based index)

- Indexed terms stored separate
- Filtering by one or both fields in any order (union/intersect as needed)

Field 1	Field 2
Lyon	France
Paris	France
Oslo	Norway

ShipTo.City Index

Index Key	Record IDs
Lyon	[7],[1],[4]
Paris	[5],[6]
Oslo	[12],[2],[9],[34]

ShipTo.Country Index

Index Key	Record IDs
France	[7],[1],[4],[5],[6]
Norway	[12],[2],[9],[34]

Collection/table scans are easily overlooked

Collection scan - development

- Small amount of data
- Extremely small query latency

Collection scan - production

- Large amount of data
- HUGE latency (quite often!)

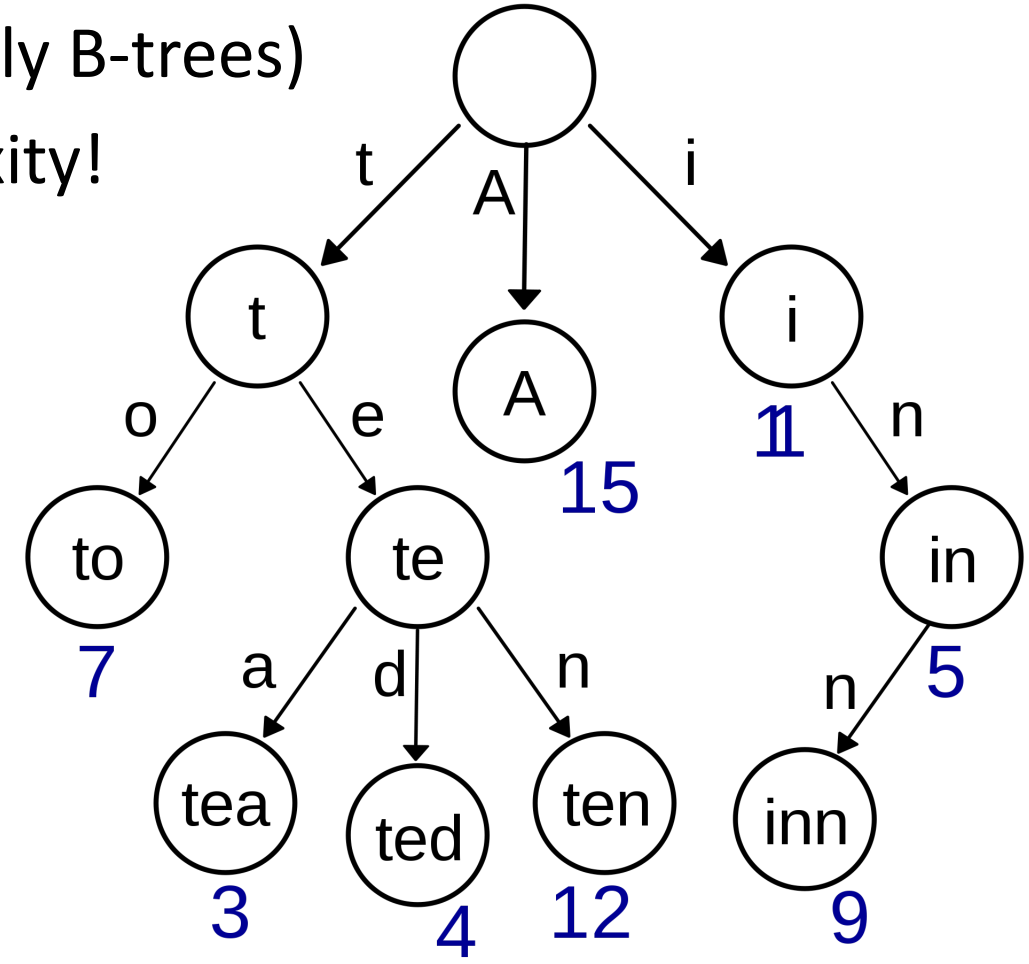
Latency: 50ms vs 50 hours

2-b) Indexing & Queries

More about indexing

Indexing

- Indexes are stored as trees (usually B-trees)
- Updates have non-trivial complexity!



https://commons.wikimedia.org/wiki/File:Trie_example.svg

Indexing

```
from Orders
where ShipTo.City in ("Paris", "Lyon") and
      OrderedAt >= '2010-05-01'
```

Search time complexity (WHERE clause):

O(log(N)) + O(log(M)) + O(Max(K,P))

Where:

- *N and M are amount of rows in indexes*
- *K and P are result sets of index searches*

And if we use RDBMS, things become even more interesting...

Join Algorithm	Complexity
Merge Join	**O(n*log(n) + m*log(m))**
Hash Join	**O(n + m)**
Index Join	**O(m*log(n))**

And if we have a non-trivial query...

```sql
SELECT
  e.employee_id AS "Employee #"
  , e.first_name || ' ' || e.last_name AS "Name"
  , e.email AS "Email"
  , e.phone_number AS "Phone"
  , TO_CHAR(e.hire_date, 'MM/DD/YYYY') AS "Hire Date"
  , TO_CHAR(e.salary, 'L99G999D99', 'NLS_NUMERIC_CHARACTERS = ''.,'' NLS_CURRENCY = ''$''') AS "Salary"
  , e.commission_pct AS "Comission %"
  , 'works as ' || j.job_title || ' in ' || d.department_name || ' department (manager: '
    || dm.first_name || ' ' || dm.last_name || ') and immediate superior: ' || m.first_name || ' ' || m.last_name AS "Current Job"
  , TO_CHAR(j.min_salary, 'L99G999D99', 'NLS_NUMERIC_CHARACTERS = ''.,'' NLS_CURRENCY = ''$''') || ' - ' ||
     TO_CHAR(j.max_salary, 'L99G999D99', 'NLS_NUMERIC_CHARACTERS = ''.,'' NLS_CURRENCY = ''$''') AS "Current Salary"
  , l.street_address || ', ' || l.postal_code || ', ' || l.city || ', ' || l.state_province || ', '
    || c.country_name || ' (' || r.region_name || ')' AS "Location"
  , jh.job_id AS "History Job ID"
  , 'worked from ' || TO_CHAR(jh.start_date, 'MM/DD/YYYY') || ' to ' || TO_CHAR(jh.end_date, 'MM/DD/YYYY') ||
    ' as ' || jj.job_title || ' in ' || dd.department_name || ' department' AS "History Job Title"

FROM employees e JOIN jobs j ON e.job_id = j.job_id
                 LEFT JOIN employees m ON e.manager_id = m.employee_id
                 LEFT JOIN departments d ON d.department_id = e.department_id
                 LEFT JOIN employees dm ON d.manager_id = dm.employee_id
                 LEFT JOIN locations l ON d.location_id = l.location_id
                 LEFT JOIN countries c ON l.country_id = c.country_id
                 LEFT JOIN regions r ON c.region_id = r.region_id
                 LEFT JOIN job_history jh ON e.employee_id = jh.employee_id
                 LEFT JOIN jobs jj ON jj.job_id = jh.job_id
                 LEFT JOIN departments dd ON dd.department_id = jh.department_id
ORDER BY e.employee_id;
```

Usually, there is also a WHERE clause, so...

Those are 10 JOIN statements!

https://dev.to/tyzia/example-of-complex-sql-query-to-get-as-much-data-as-possible-from-database-9he

…we have complexity between *O(log(n))* and *O(too much)*!

More often than not it is O(too much)…

What can (should!) we do?

- RDBMS
  - Proper indexing (kinda obvious, but still ☺)
  - Optimize (remove unnecessary JOINs – depends on business logic)
  - Reduce query complexity
    - Replace ‘row by row’ cursors with set based queries
    - Reduce the amount of work queries do (for example, unnecessary sub-queries)
    - Remove ORDER BY where it makes sense (huge overhead)
    - Other optimizations are possible
- NoSQL
  - Proper modeling
  - Well planned indexing

2-c) Indexing & Queries

Indexes (sometimes) have complexity too!

Indexing complexity

```
db.runCommand( {
    mapReduce: "blogs",
    map: function(){
        for (let index = 0; index < this.authors.length; ++index) {
            let author = this.authors[ index ];
            emit( author.firstName + " " + author.lastName, 1 );
        }
    },
    reduce: function(author, counters){
        count = 0;

        for (let index = 0; index < counters.length; ++index) {
            count += counters[index];
        }

        return count;
    },
    out: { inline: 1 }
} )
```

Indexing complexity

Maps

Syntax ?

```
from order in docs.Orders
select new
{
    order.Company,
    Count = 1,
    Total = order.Lines.Sum(l => (l.Quantity * l.PricePerUnit) * (1 - l.Discount))
}
```

Press Shift+F1

+ Add map

Reduce

Syntax ?

```
from result in results
group result by result.Company
into g
select new
{
    Company = g.Key,
    Count = g.Sum(x => x.Count),
    Total = g.Sum(x => x.Total)
}
```

Press Shift+F1

Indexing complexity

- As you can see, indexing has its own complexity
- More often than not it can be optimized

3-a) Network

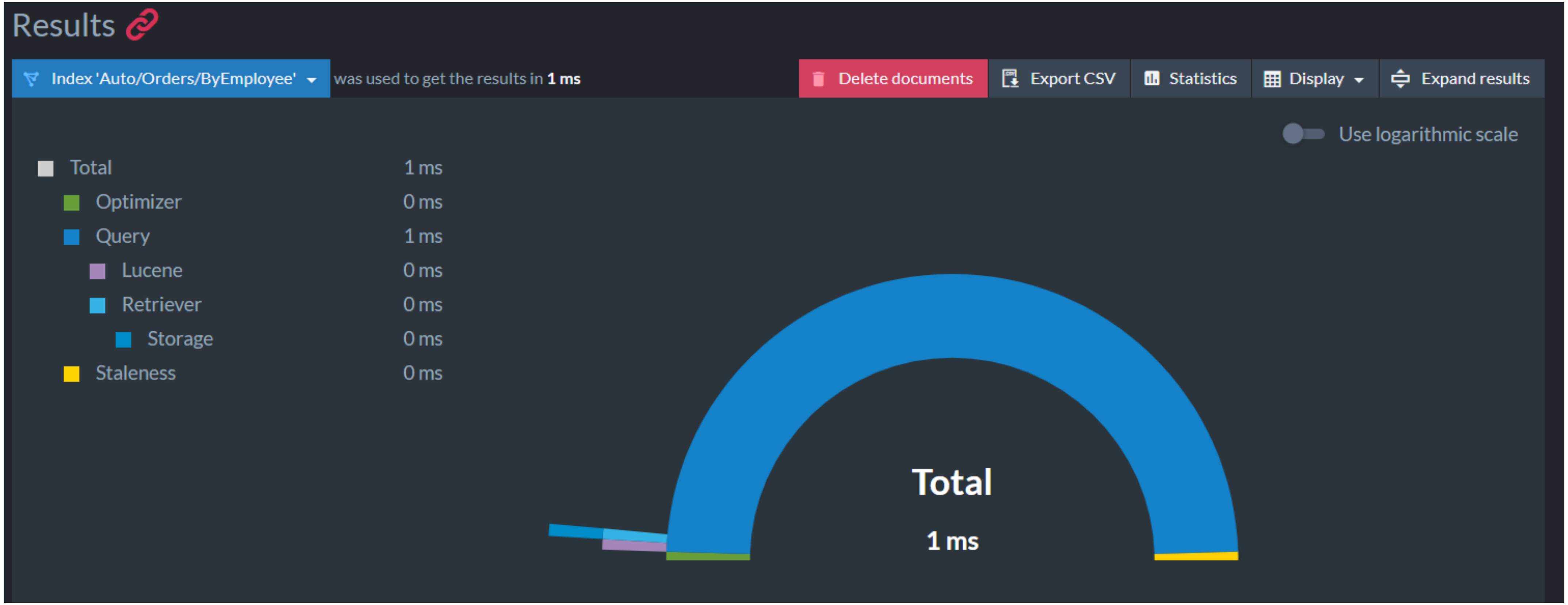
Distributed system fallacy: Bandwidth is infinite

Here is a riddle: why a query with 100 results takes several seconds to complete?

Hint: the request spends < 10ms on the server

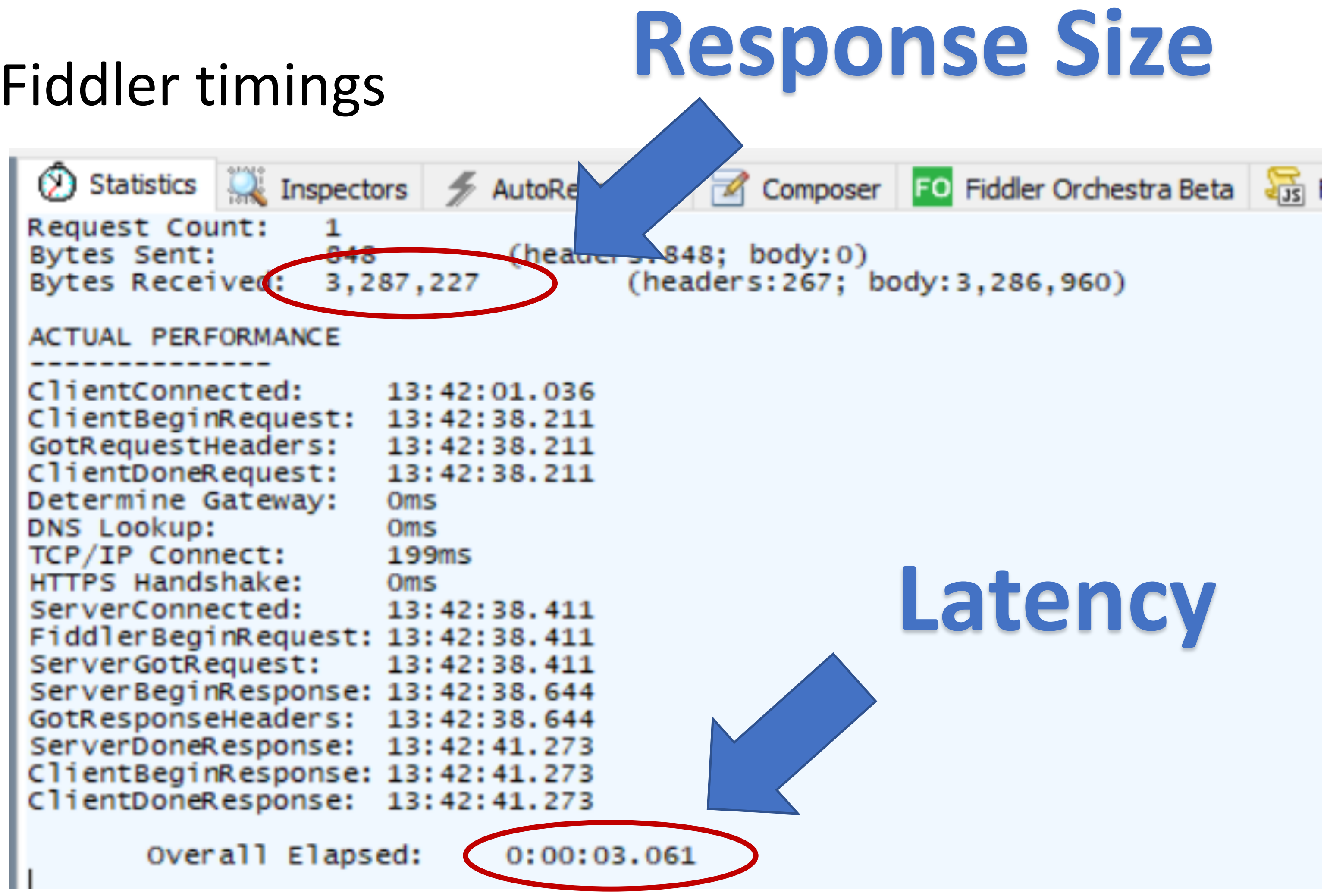
The investigation

1. Look at query latency on the server

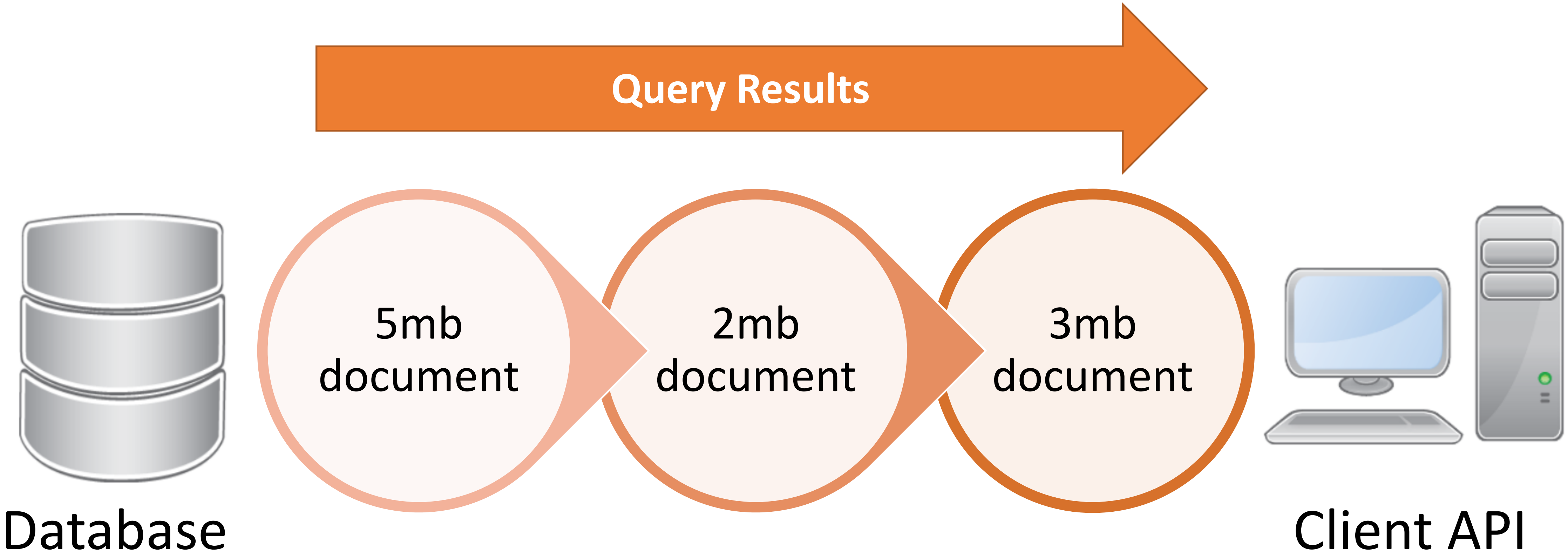


The investigation

2. Look at Fiddler timings



Network bandwith is not infinite!



Solution: server-side projections (NoSQL)

RQL

```
from index 'Orders/ByShipment/Location'
where spatial.within(ShipmentLocation,
                spatial.wkt('Circle(2.349014 48.864716 d=1000.0000)'))
load Company as c, Employee as e
select {
    CompanyName: c.Name,
    EmployeeName: e.FirstName + " " + e.LastName
}
```

Server-side Projection

Solution: server-side projections (NoSQL)

MongoDB API

```
var findDocuments = function(db, callback) {
  var collection = db.collection( 'restaurants' );
  // Find some documents
  collection.find({ 'cuisine' : 'Brazilian' }, { 'name' : 1, 'cuisine' : 1 })
          .toArray(function(err, docs) {
              console.log("Found the following records");
              //do something with found records
          });
}
```

Server-side Projection

3-b) Network

Distributed system fallacy: Latency is zero

Also… database requests can be an interesting issue…

Support » Plugin: WooCommerce » Site being slowed by too many (5000+) database queries

Site being slowed by too many (5000+) database queries

dynit (@dynit)
3 months, 3 weeks ago

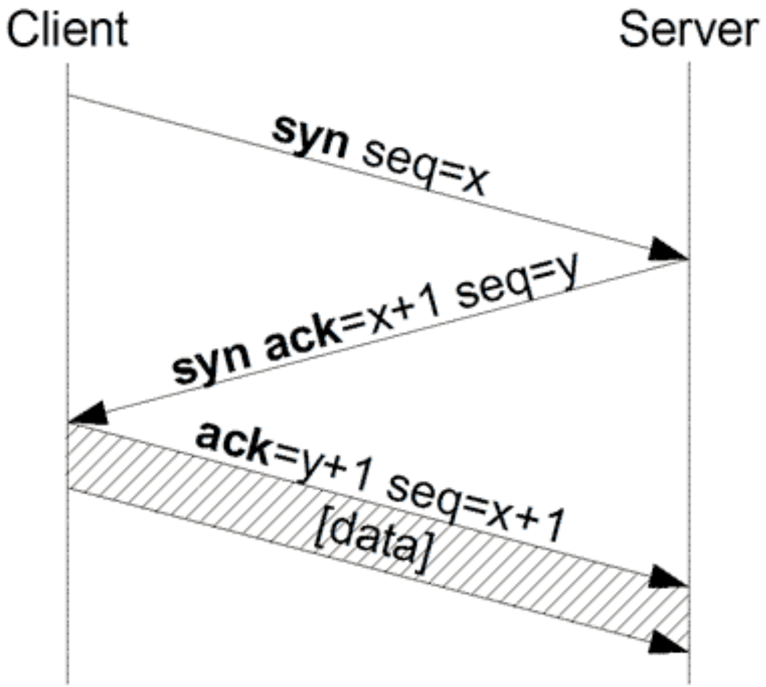
✓ Resolved

Hello,

One of our sites is really getting slowed with too many Woocommerce database queries. I checked via Query Monitor and just browsing through categories (on the default shop page with the post type archive for products) fires 5000+ database queries per page load. The site has 66 categories, about 600 products, 2200 variations, and 6 custom attributes per product. Single product pages are 'only' 150-300 queries each, no performance issue there.

Network overhead

TCP handshake




```
Route.Add("/hello",
    (request, response, args) =>
        response.AsText("world!"));

var cts = new CancellationTokenSource();
var _ = HttpServer.ListenAsync(8080,
            cts.Token, Route.OnHttpRequestAsync);

Console.ReadKey();

cts.Cancel();
```

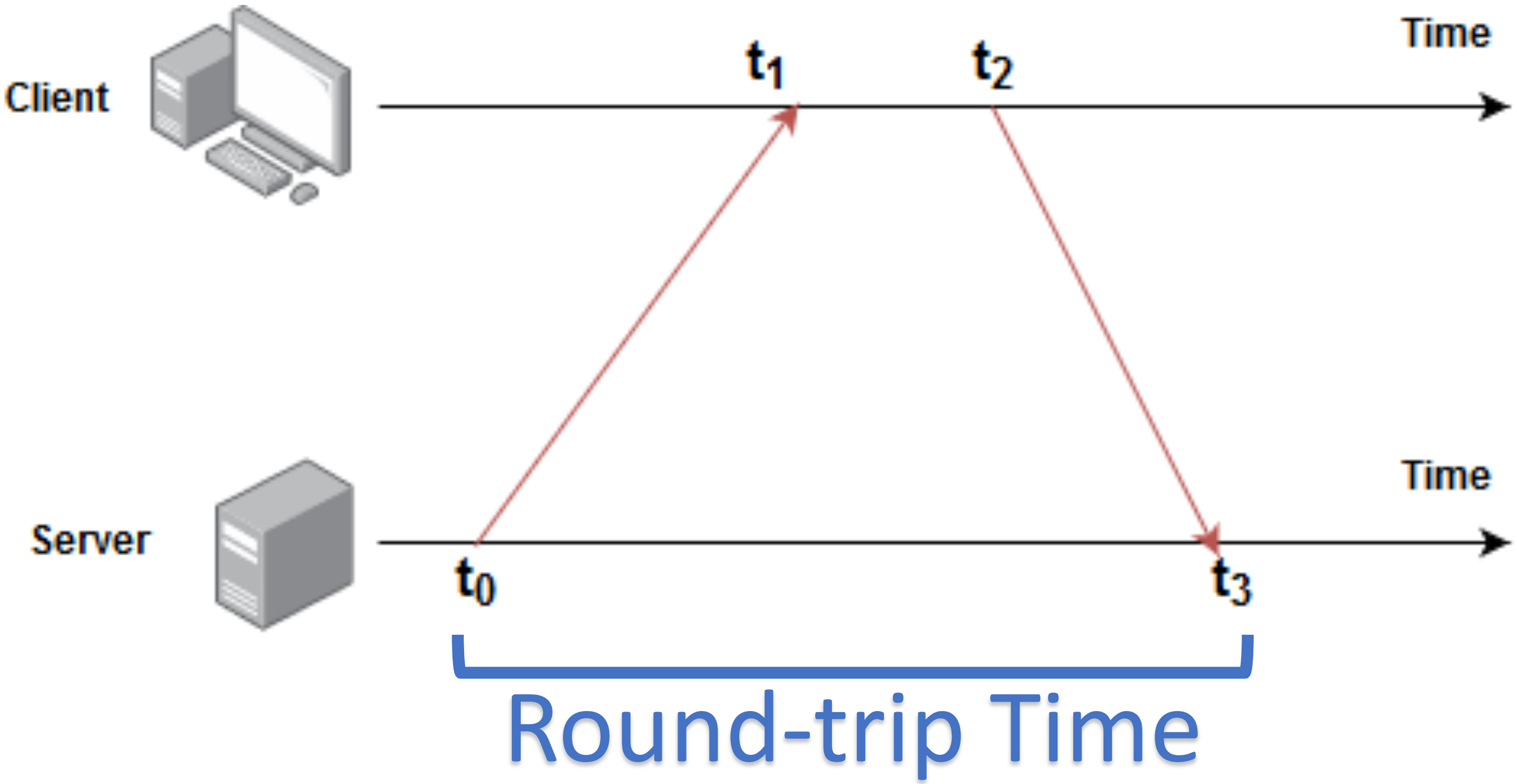
```
michael@michael-pc:~$ wrk -t16 -c1000 -d30s http://127.0.0.1:8080/hello
Running 30s test @ http://127.0.0.1:8080/hello
  16 threads and 1000 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency    21.62ms   [illegible].93ms  61.91ms   93.59%
    Req/Sec     2.87k   290.86    18.69k    93.98%
  1369876 requests in 30.09s, 178.98MB read
Requests/sec:  45524.71
Transfer/sec:      5.95MB
```

https://github.com/dajuric/simple-http

Network overhead

Round Trip Time (RTT)

- Physical distance (insignificant for LANs)
- Bandwidth
- Network hops



What can we do?

- Refactor to reduce number of requests (kinda obvious, but still…)
- NHibernate – Future Queries
- Entity Framework - QueryFuture
- RavenDB – Lazy Queries

```
using(var session = store.OpenSession())
{
    var lazyUser = session.Advanced.Lazily.Load<User>("users/michael");
    var lazyPosts = session.Query<Posts>().Take(30).Lazily();

    session.Advanced.Eagerly.ExecuteAllPendingLazyOperations();

    //do something with lazyUser and lazyPosts
}
```

May sound trivial, but…

*Do take a look at database traffic while **stress testing** and if possible in **production** too.*

- Fiddler
- Wireshark
- Profilers
- Any other tool to inspect traffic

To sum it up

- Databases are abstractions
- Abstractions are leaky and might be the cause of perf issues
- Such perf issues can be dealt with (if we know about the "leak"!)

Questions?

michael.yarichuk@hibernatingrhinos.com

@myarichuk

https://github.com/ravendb/ravendb

https://github.com/myarichuk/PerfDemo-Sequential-vs-Random-Key