The Power of the Log

LSM & Append Only Data Structures

Ben Stopford

Confluent Inc

confluent

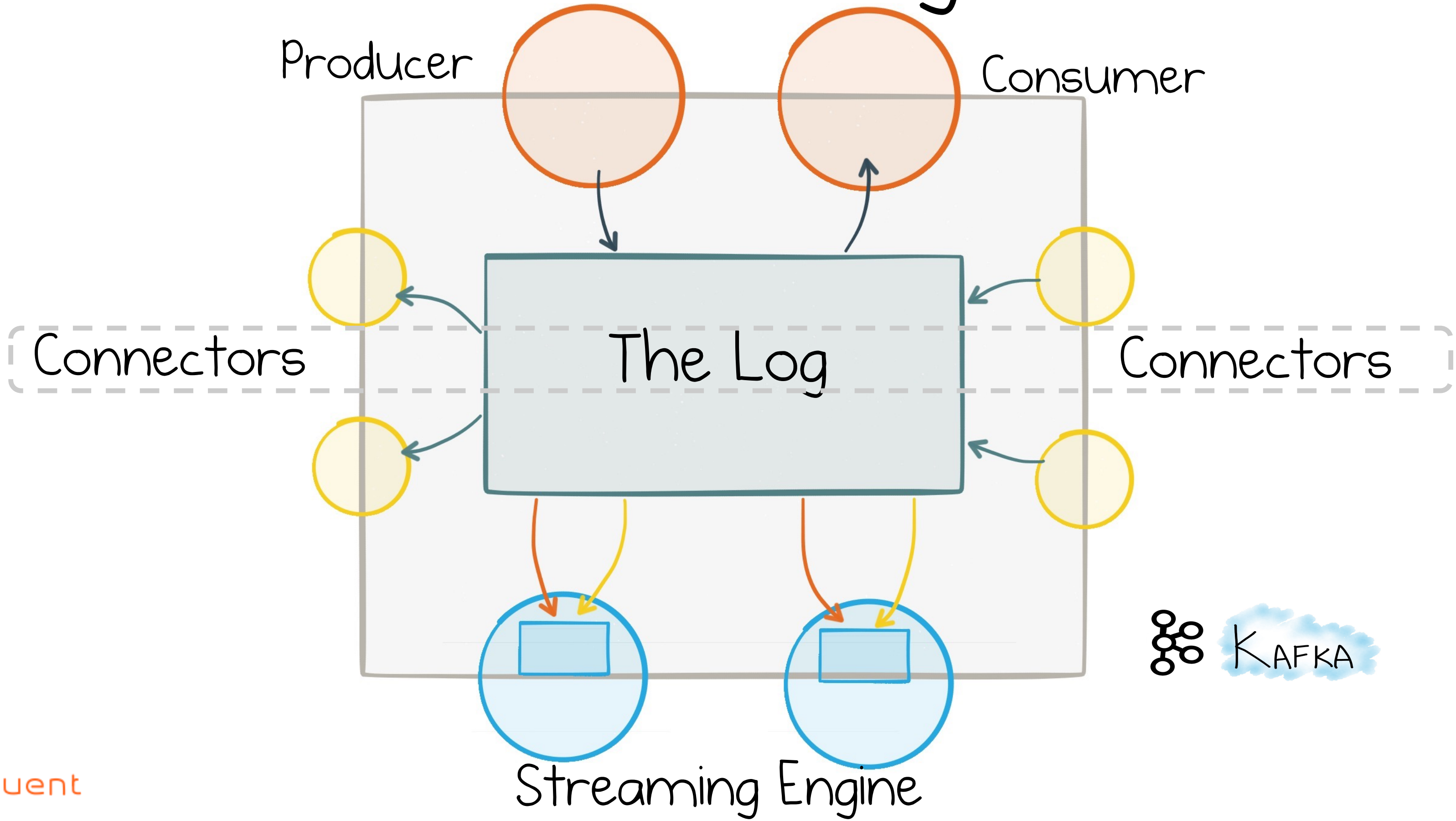
THIS IS ME

- ENGINEER AT CONFLUENT
- EX THOUGHTWORKS + UK FINANCE

@benstopford

confluent

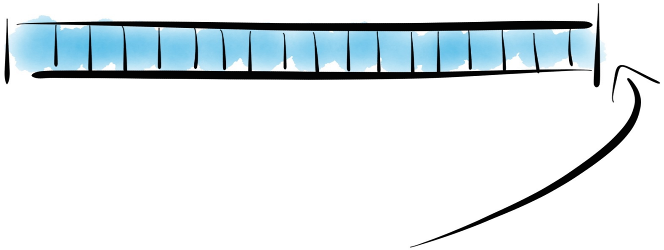
Kafka: a Streaming Platform



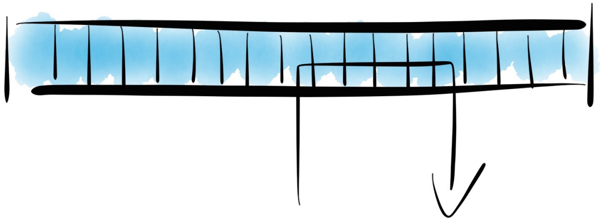
KAFKA

confluent

KAFKA's Distributed Log

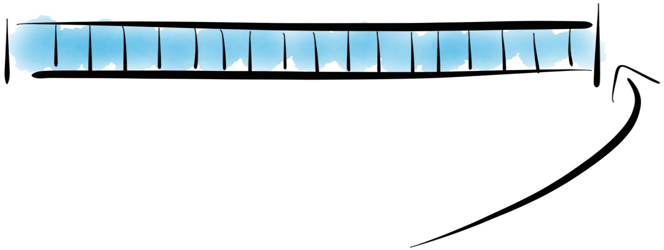


Append Only

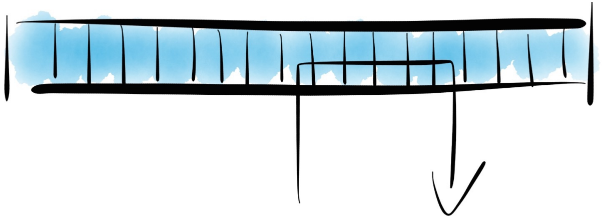


Linear Scans

Messaging is a Log-Shaped Problem



Append Only



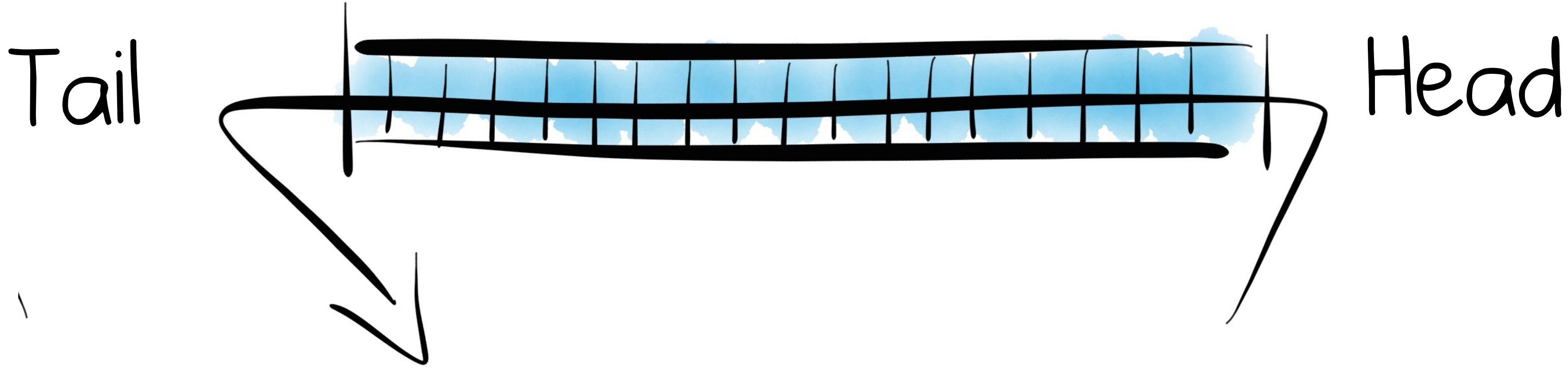
Linear Scans

Not all problems are Log-Shaped

Many problems benefit from being addressed in a “log-shaped” way

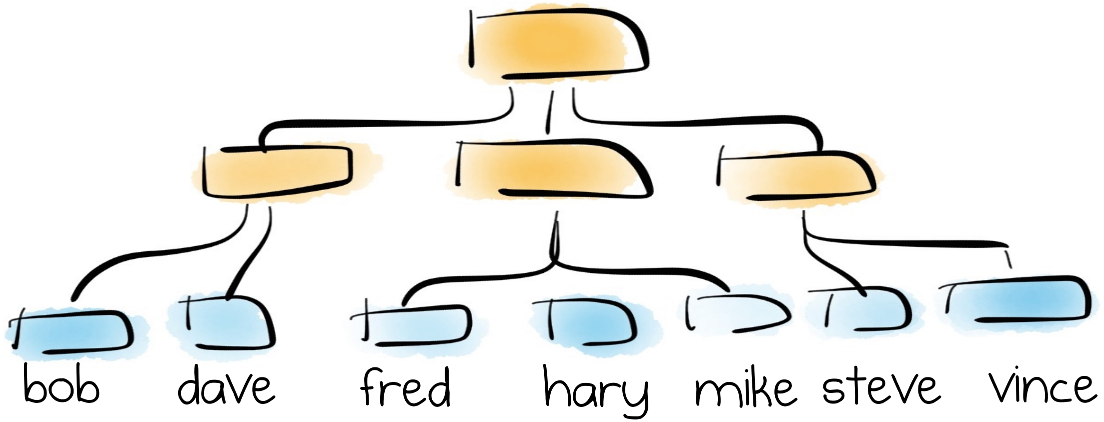
Supporting Lookups

Lookups in a log



Trees provide Selectivity

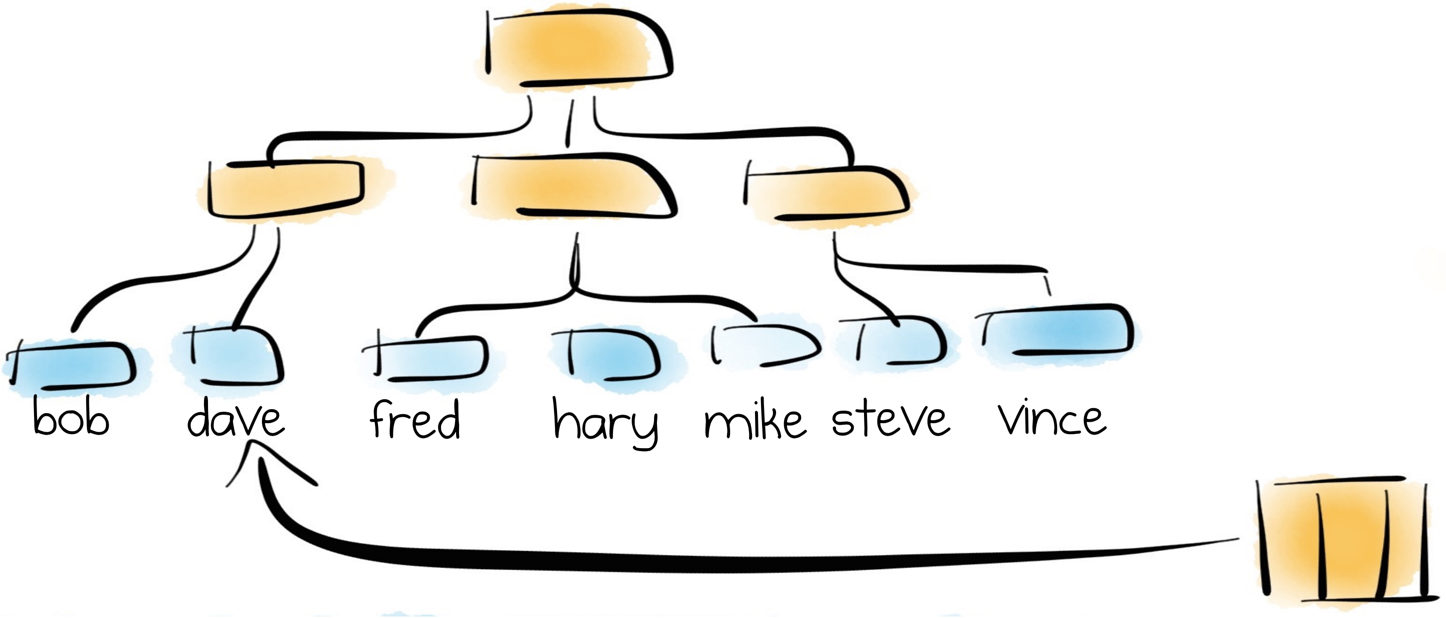
Index



confluent

But the overarching structure implies Dispersed Writes

Random IO



confluent

Log Structured Merge Trees

1996

The Log-Structured Merge-Tree (LSM-Tree)

Patrick O'Neil[1], Edward Cheng[2]
Dieter Gawlick[3], Elizabeth O'Neil[1]
To be published: Acta Informatica

ABSTRACT. High-performance transaction system applications typically insert rows in a History table to provide an activity trace; at the same time the transaction system generates log records for purposes of system recovery. Both types of generated information can benefit from efficient indexing. An example in a well-known setting is the TPC-A benchmark application, modified to support efficient queries on the History for account activity for specific accounts. This requires an index by account-id on the fast-growing History table. Unfortunately, standard disk-based index structures such as the B-tree will effectively double the I/O cost of the transaction to maintain an index such as this in real time, increasing the total system cost up to fifty percent. Clearly a method for maintaining a real-time index at low cost is desirable. The Log-Structured Merge-tree (LSM-tree) is a disk-based data structure designed to provide low-cost indexing for a file experiencing a high rate of record inserts (and deletes) over an extended period. The LSM-tree uses an algorithm that defers and batches index changes, cascading the changes from a memory-based component through one or more disk components in an efficient manner reminiscent of merge sort. During this process all index values are continuously accessible to retrievals (aside from very short locking periods), either through the memory component or one of the disk components. The algorithm has greatly reduced disk arm movements compared to a traditional access methods such as B-trees, and will improve cost-performance in domains where disk arm costs for inserts with traditional access methods

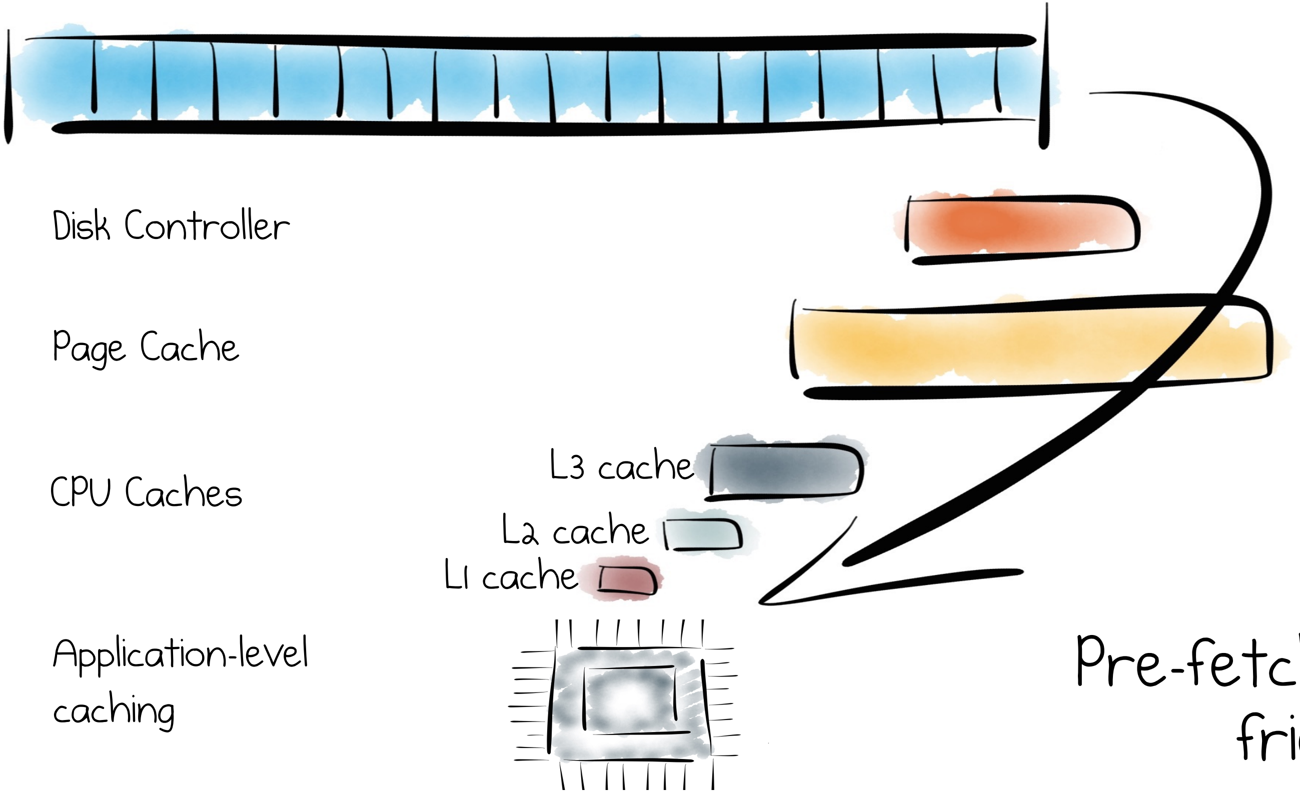
Used in a range of modern databases

- BigTable
- HBase
- LevelDB
- SQLite4
- RocksDB
- MongoDB
- WiredTiger
- Cassandra
- MySQL
- InfluxDB ...

If a systems have a natural grain, it is one formed of sequential operations which favour locality

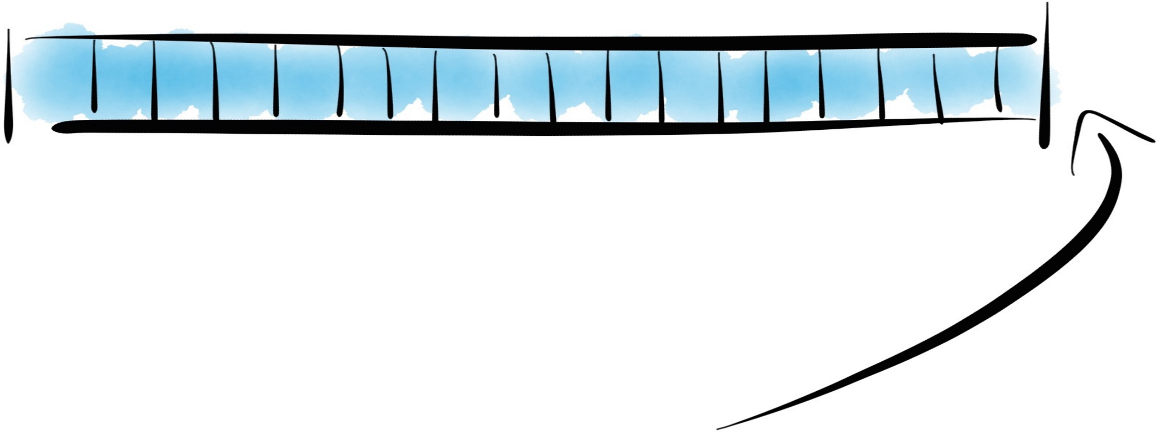
confluent

Caching & Prefetching

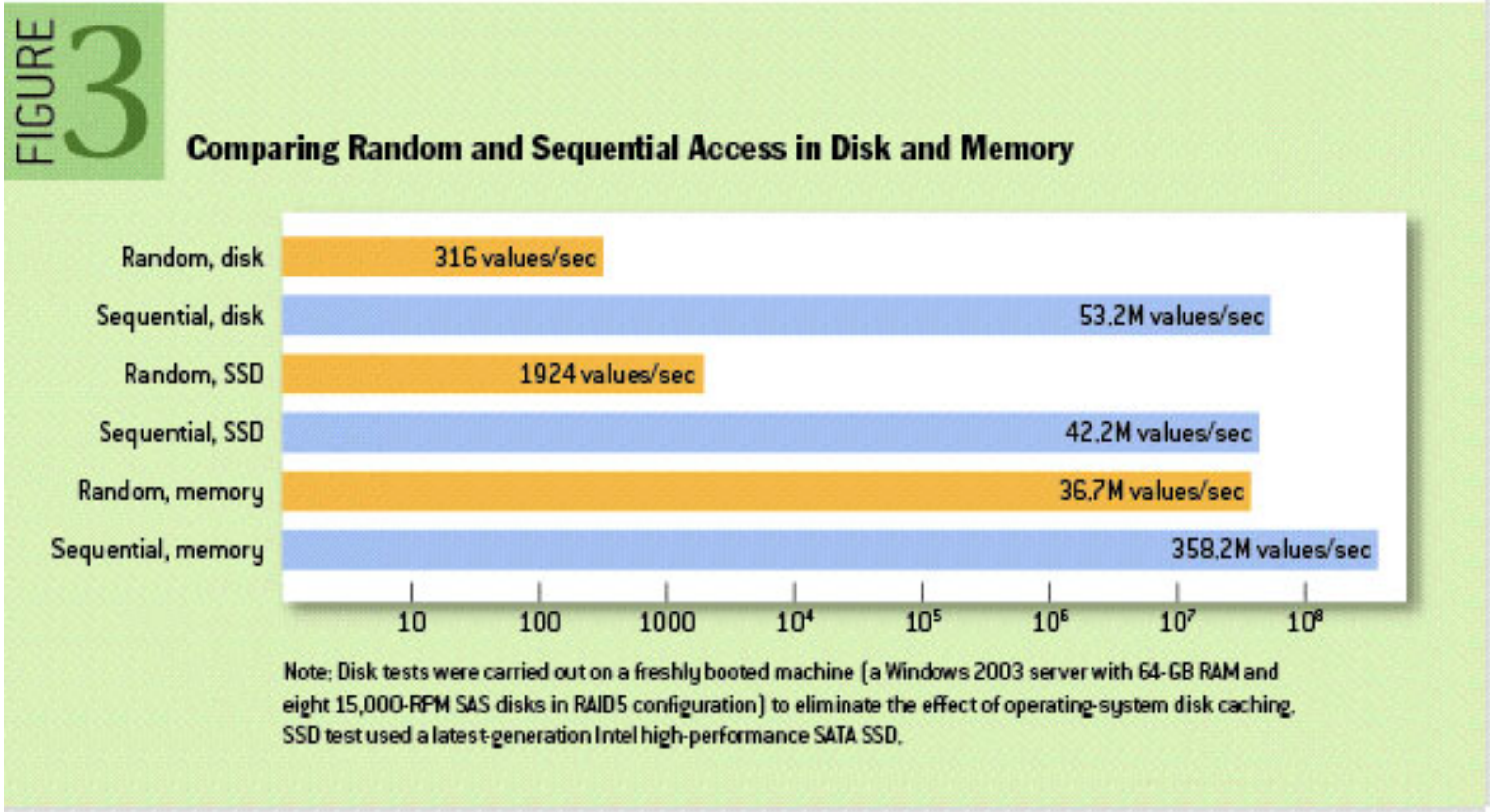


Pre-fetch is your friend

Write efficiency comes from amortising writes into sequential operations



confluent

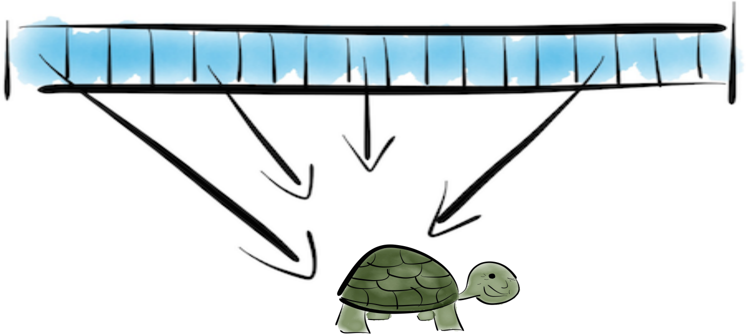


confluent

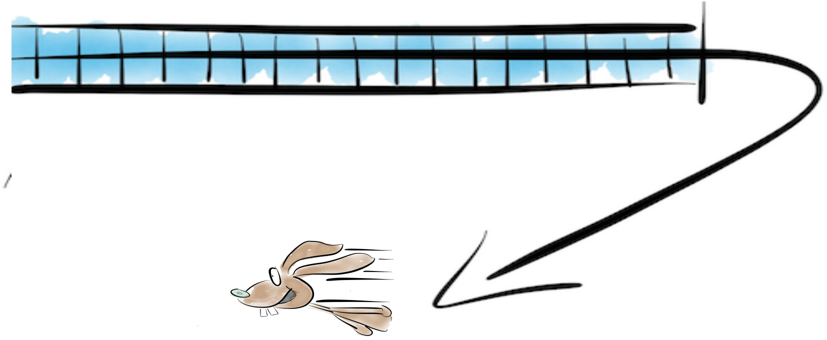
Taken from ACMQueue: The Pathologies of Big Data

So if we go against the grain of the system, RAM can actually be slower than disk

Going against the grain means dispersed operations that break locality



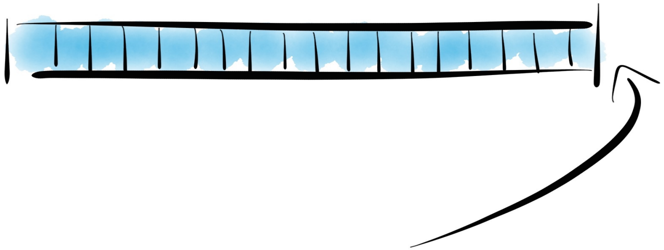
Poor Locality



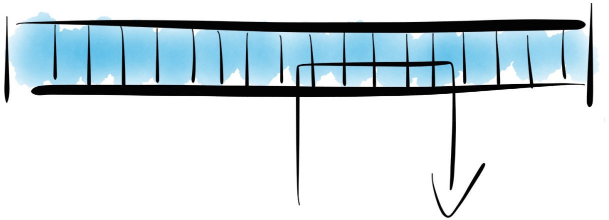
Good Locality

confluent

The beauty of the log lies in its sequentially



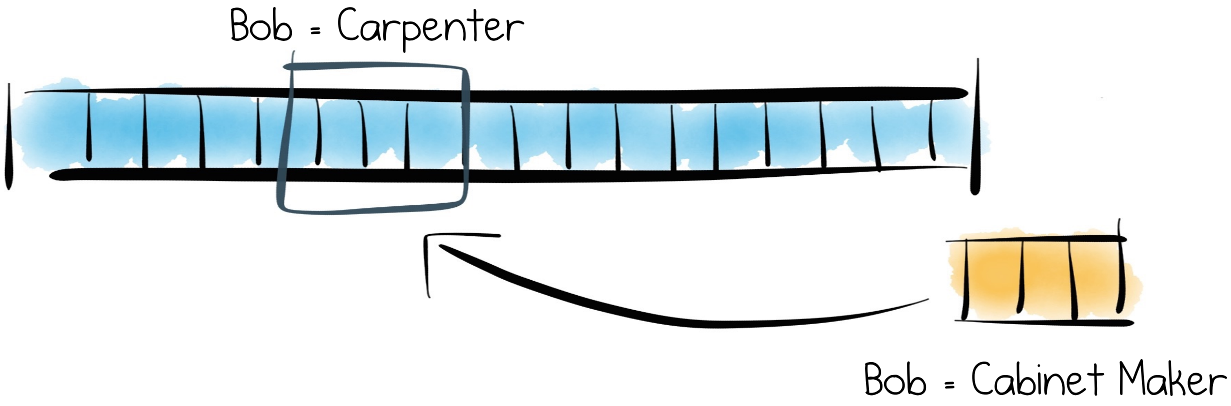
Append Only



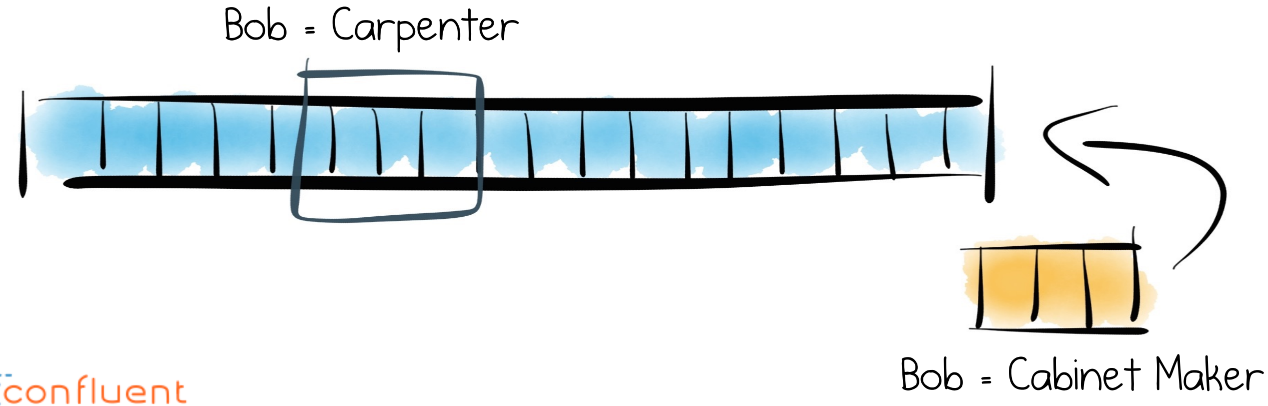
Linear Scans

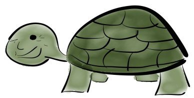
LSM is about re-imagining search as as a “log-shaped” problem

Arrange writes to be Append Only



Update in Place
Ordered File
(Random IO)

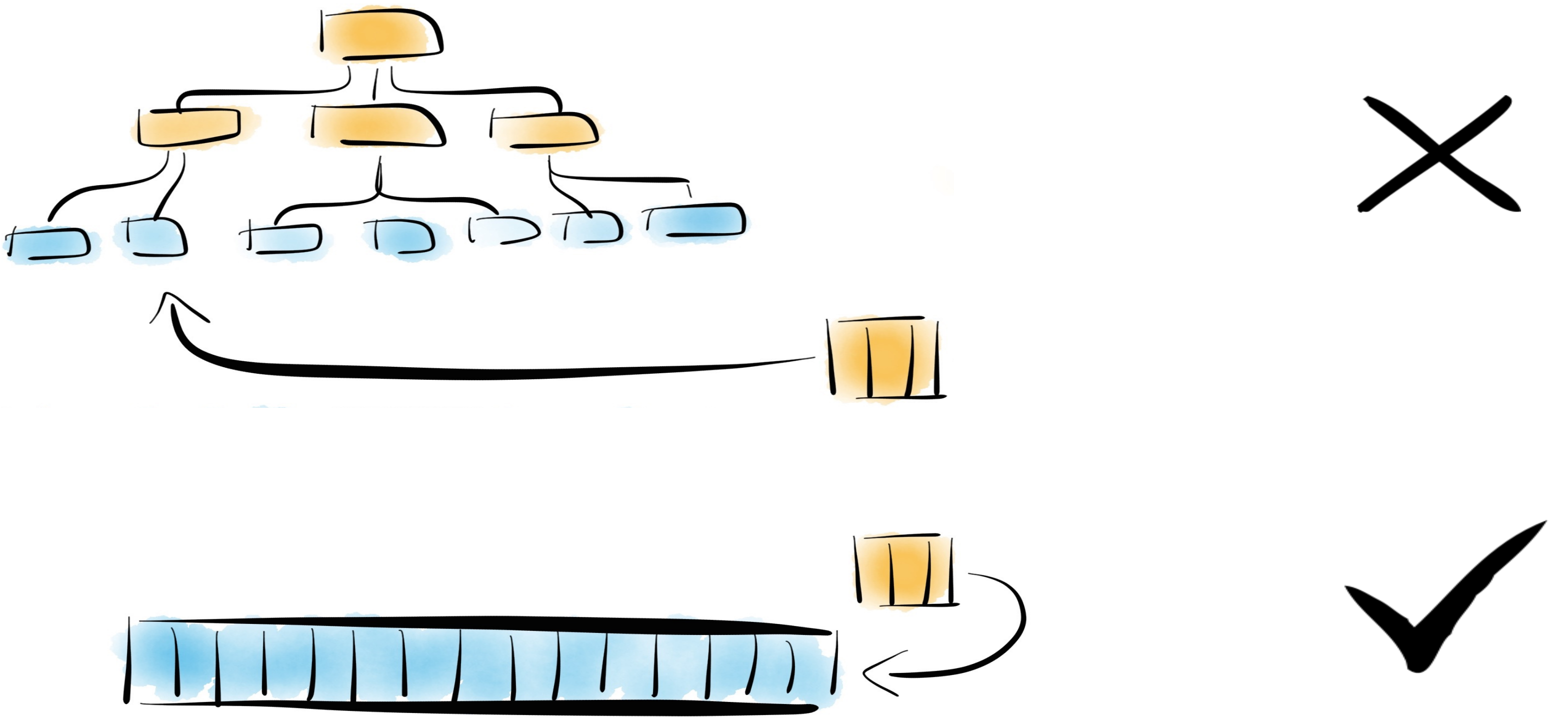




Append Only
Journal
(Sequential IO)

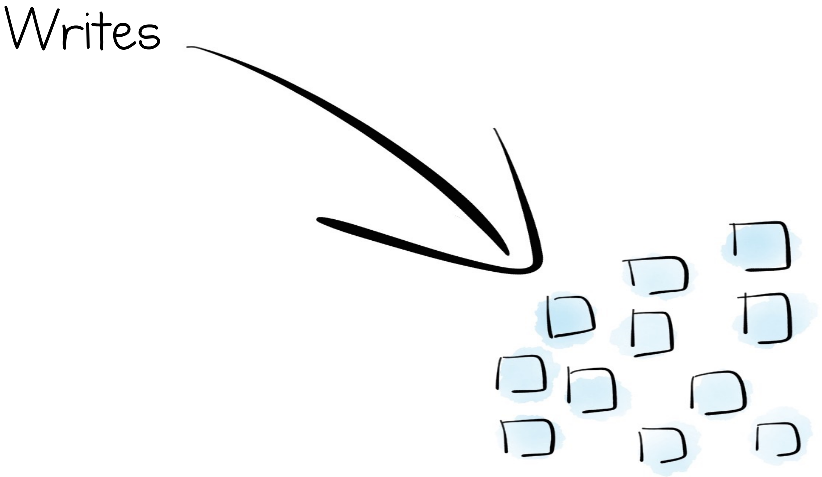
confluent

Avoid dispersed writes



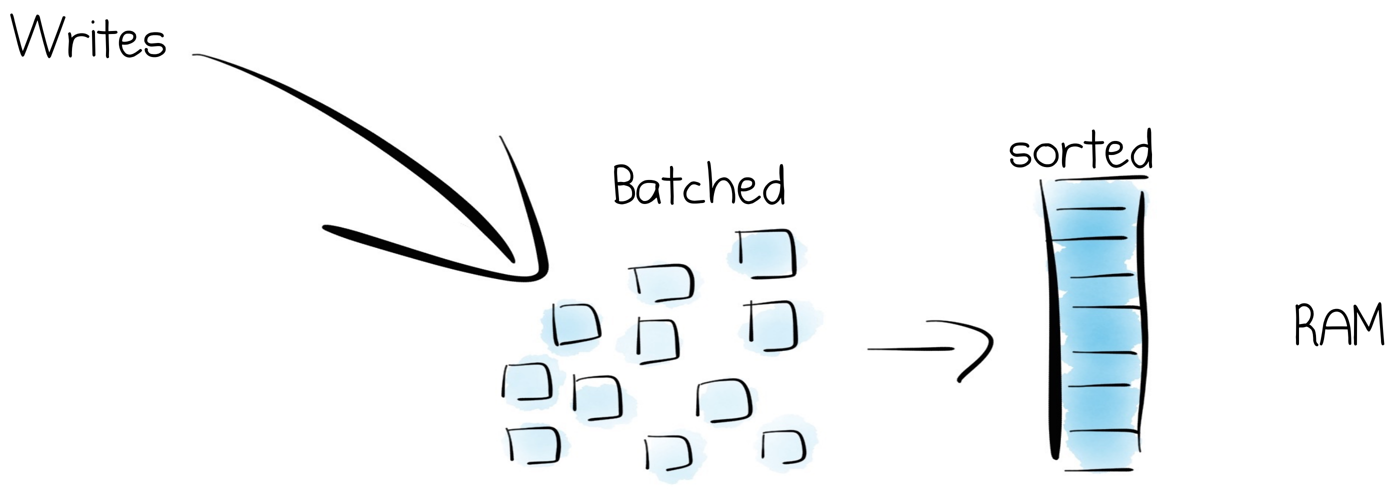
Simple LSM

Writes are collected in memory

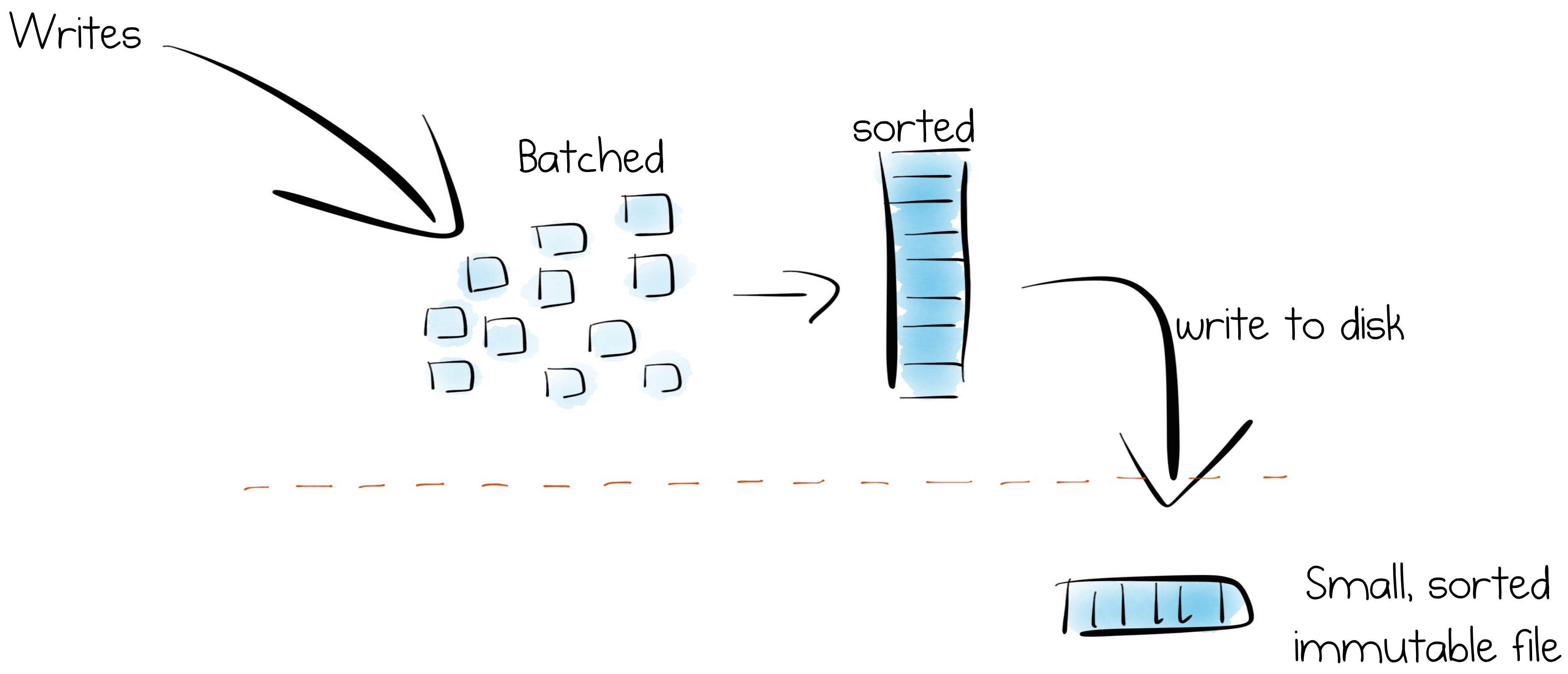


RAM

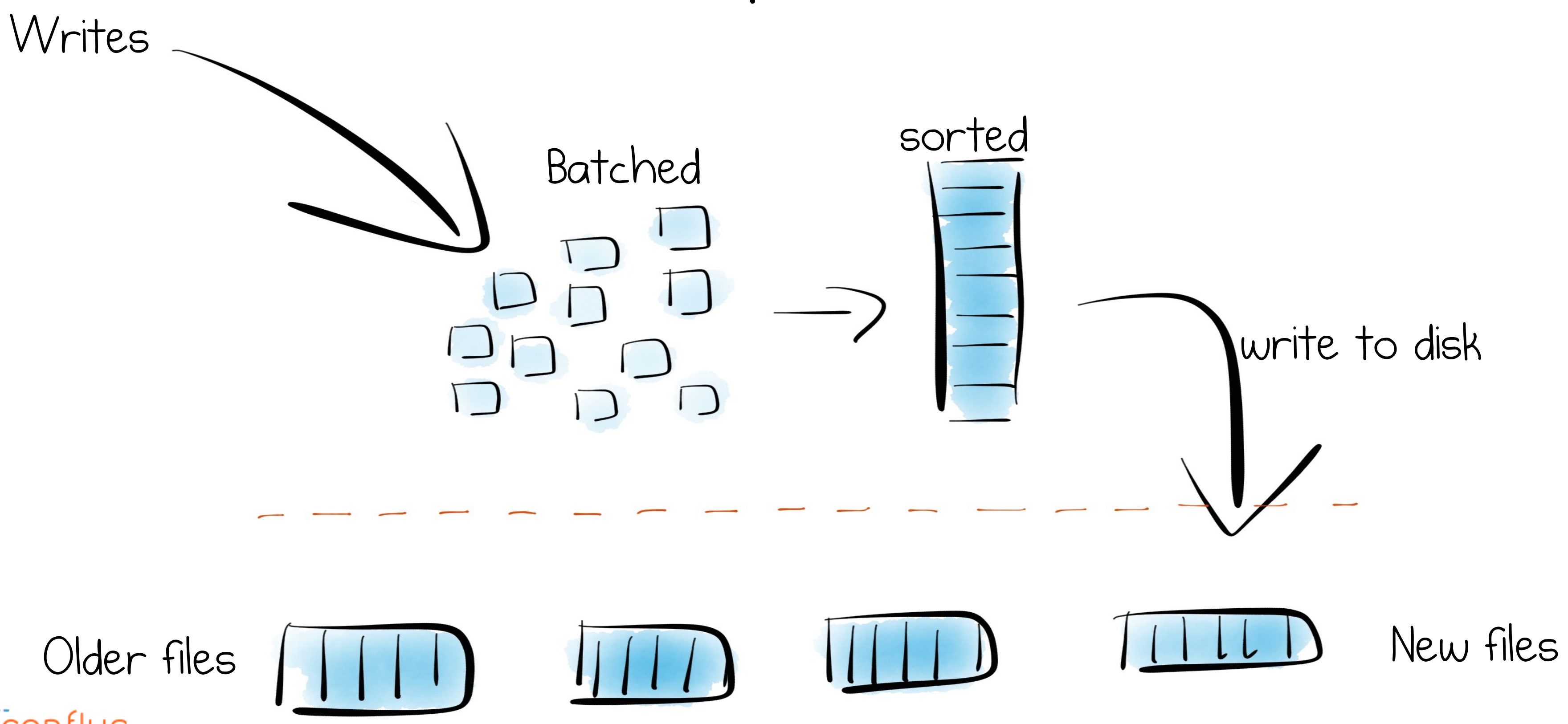
When enough have buffered, sort.



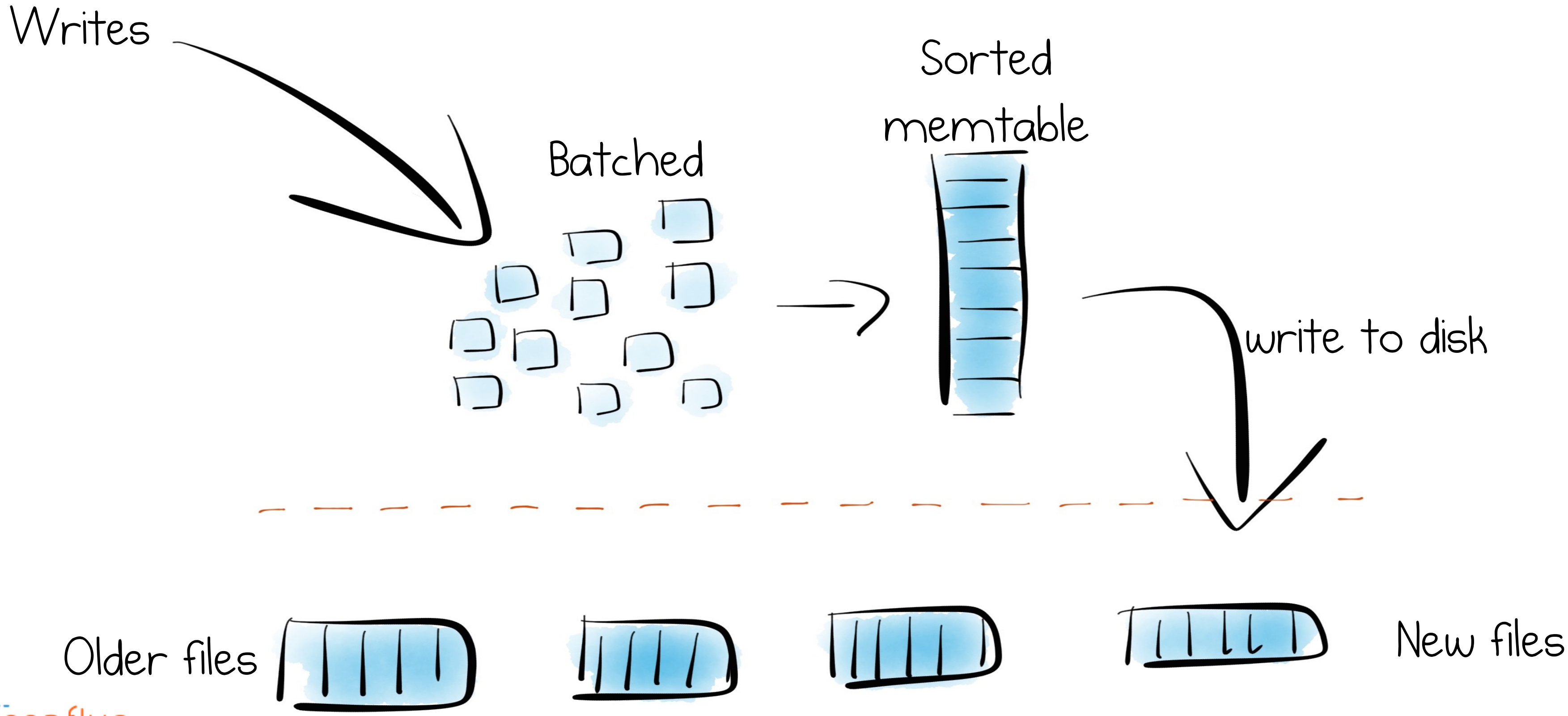
Write the sorted file to disk



Repeat...



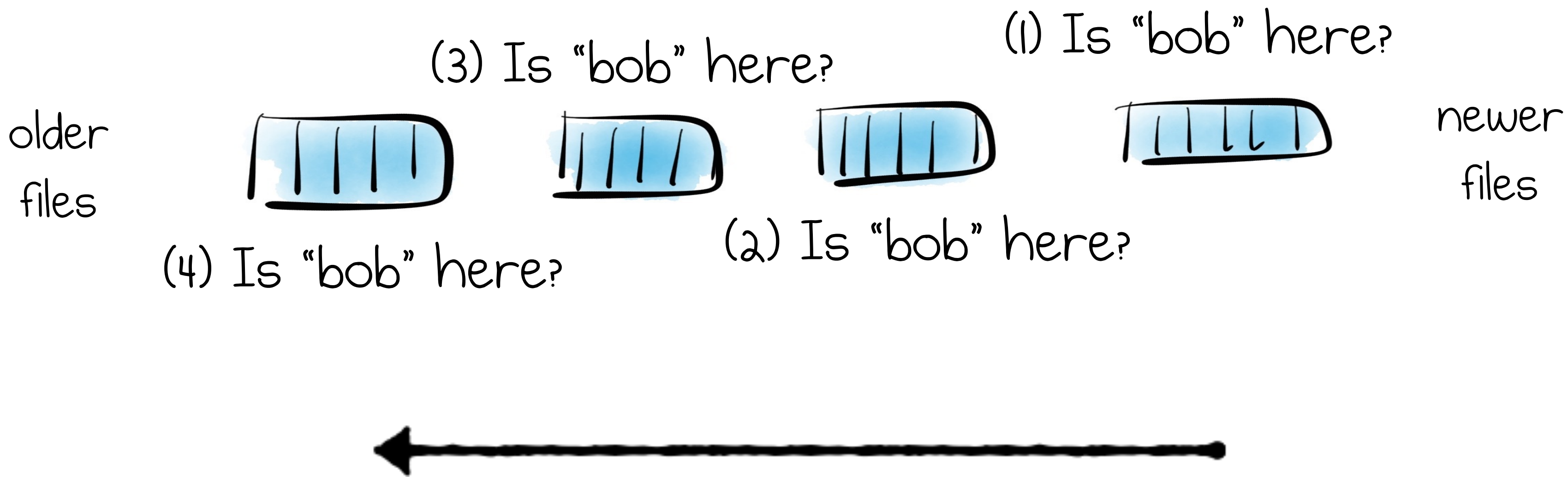
Batching -> Fast Sequential IO



That's the core write path

What about reads?

Search reverse-chronologically

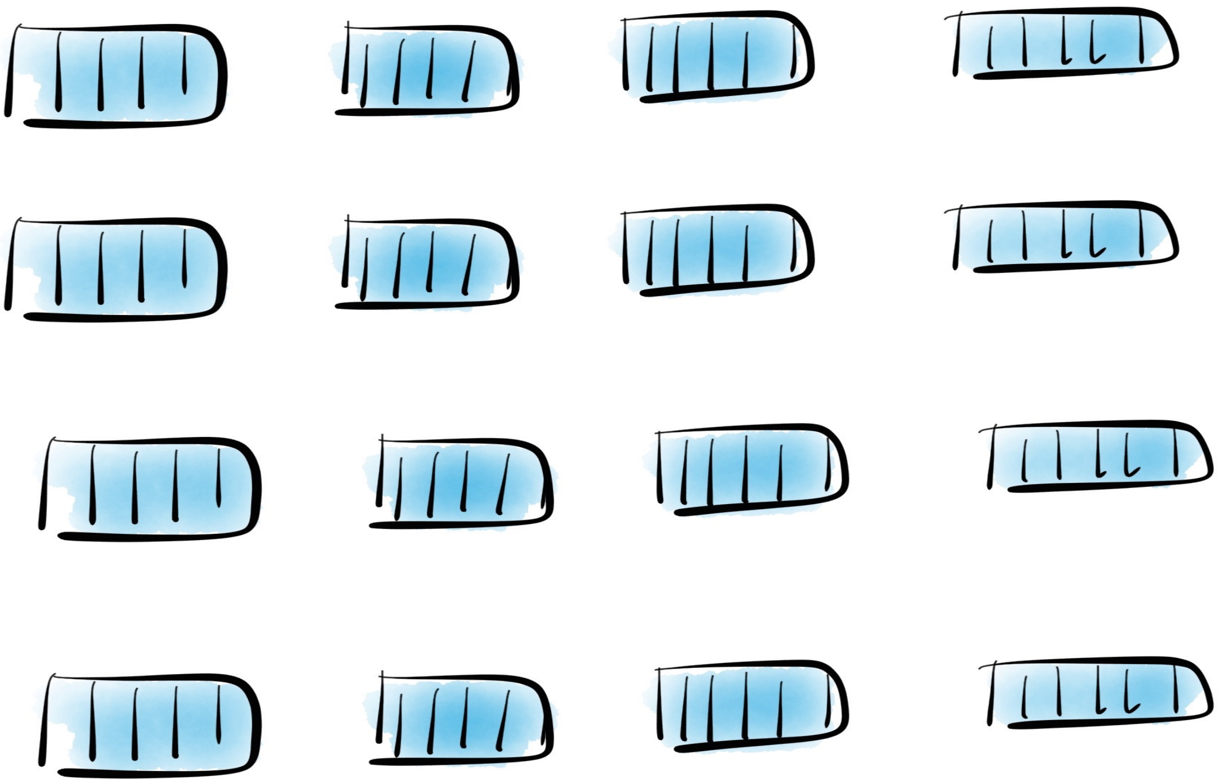


confluent

Worst Case

We consult every file

We might have a lot of files!



LSM naturally optimises for writes, over reads

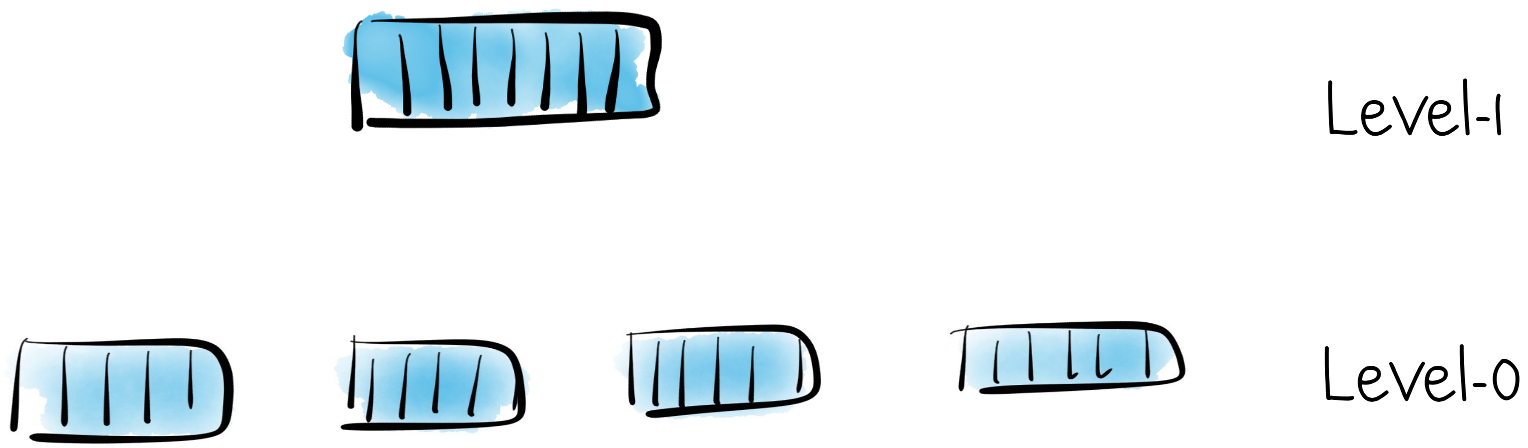
This is a reasonable tradeoff to make

Optimizing reads is easier than optimising writes

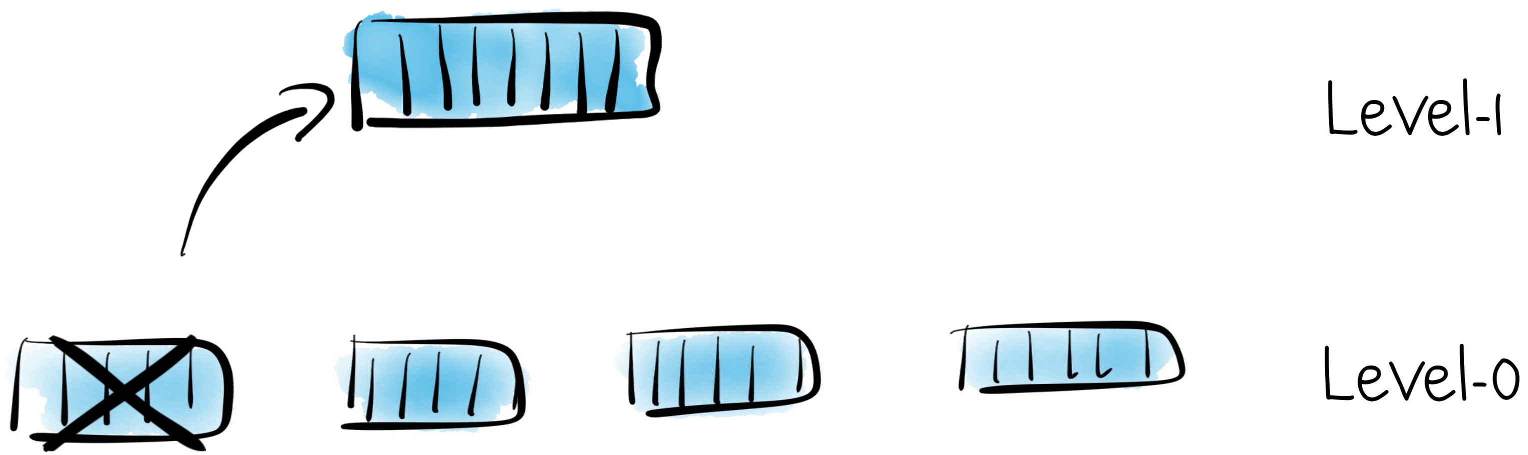
Optimisation 1

Bound the number of files

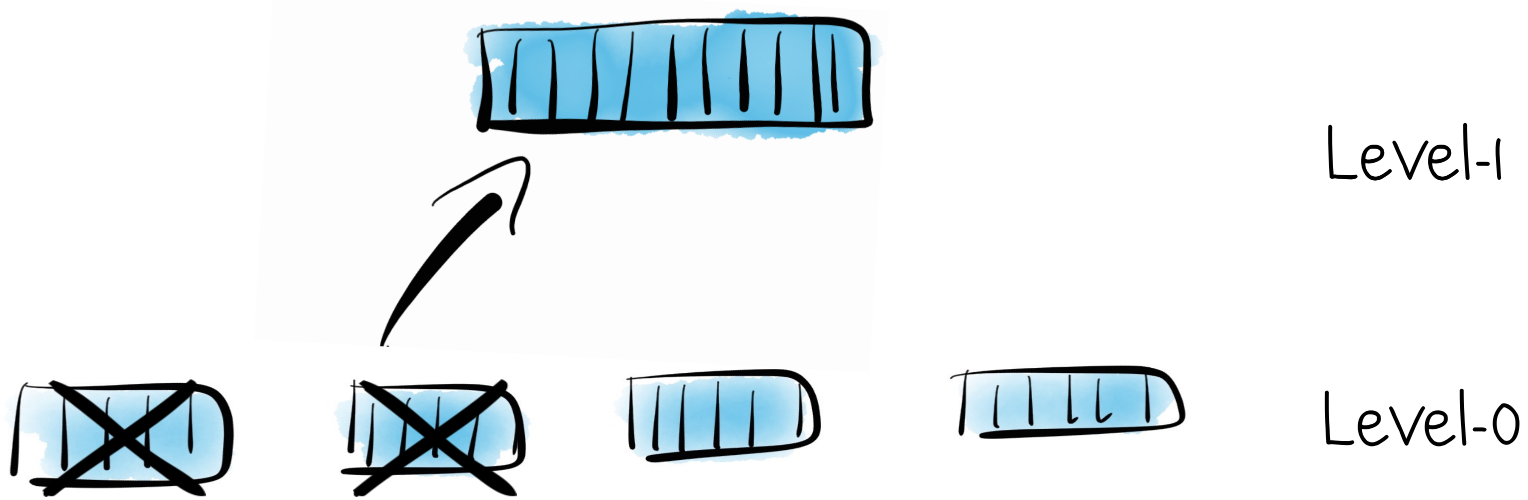
Create levels



Separate thread merges old files, de-duplicating them.

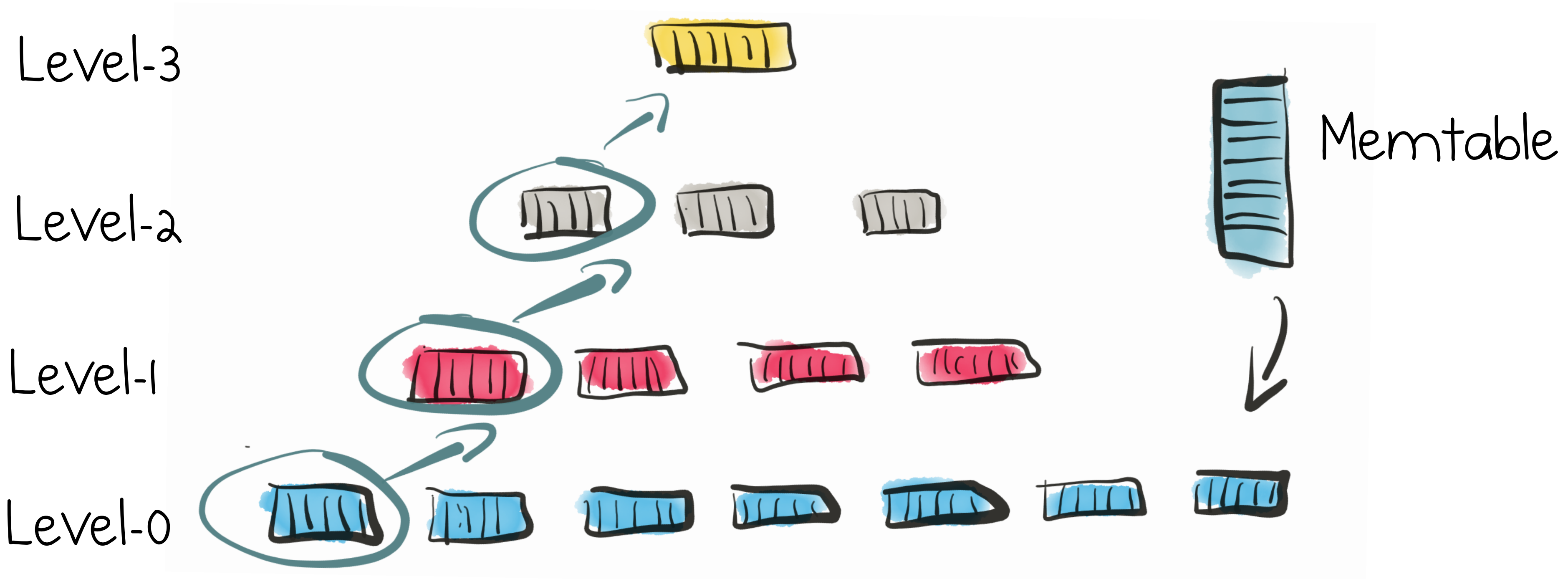


Separate thread merges old files, de-duplicating them.



Merging process is reminiscent of merge sort

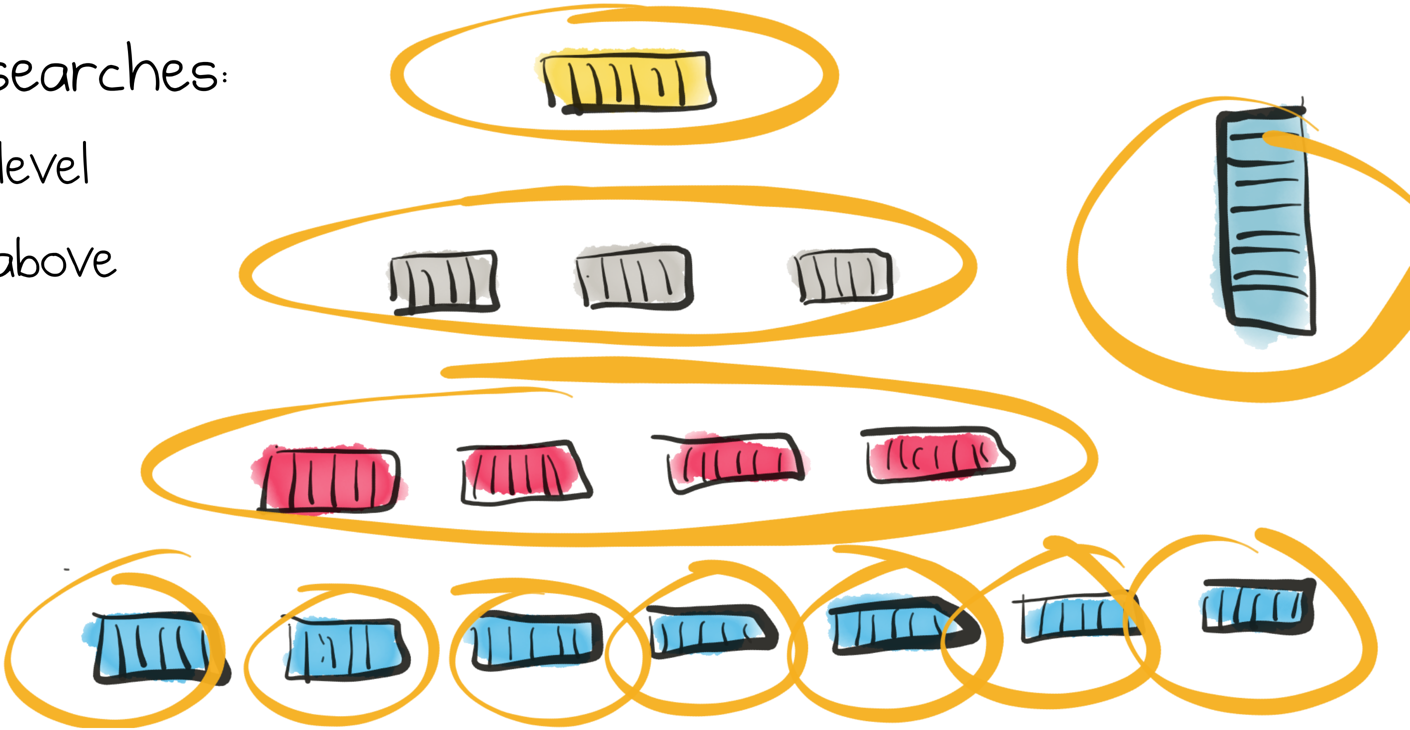
Take this further with levels



confluent

But single reads still require many individual lookups:

- Number of searches:
  - 1 per base level
  - 1 per level above



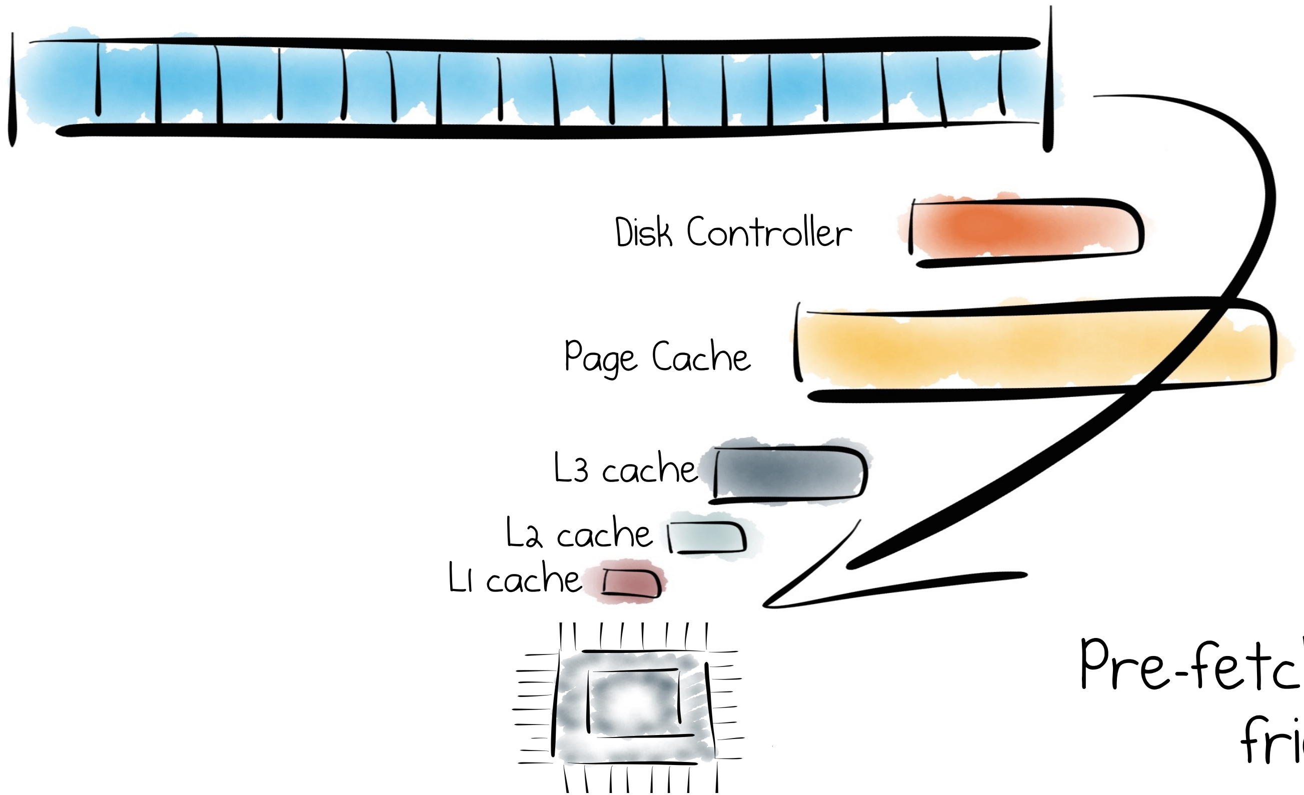
Optimisation 2

Caching & Friends

Add Memory

i.e. More Caching / Pre-fetch

Read Ahead & Prefetch



Pre-fetch is your friend

If only there was a more efficient way to avoid searching each file!

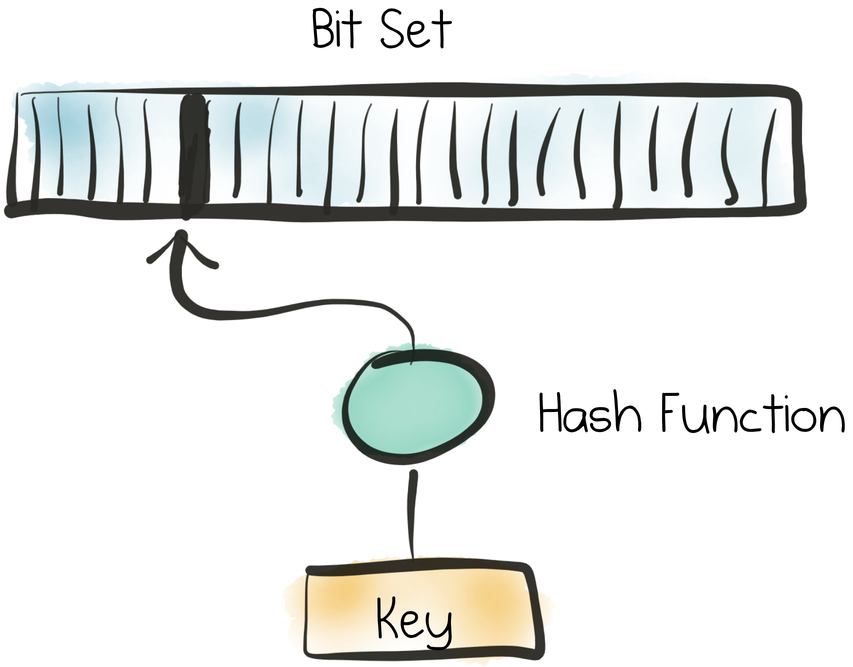
Elven Magic?

Bloom Filters

Answers the question:

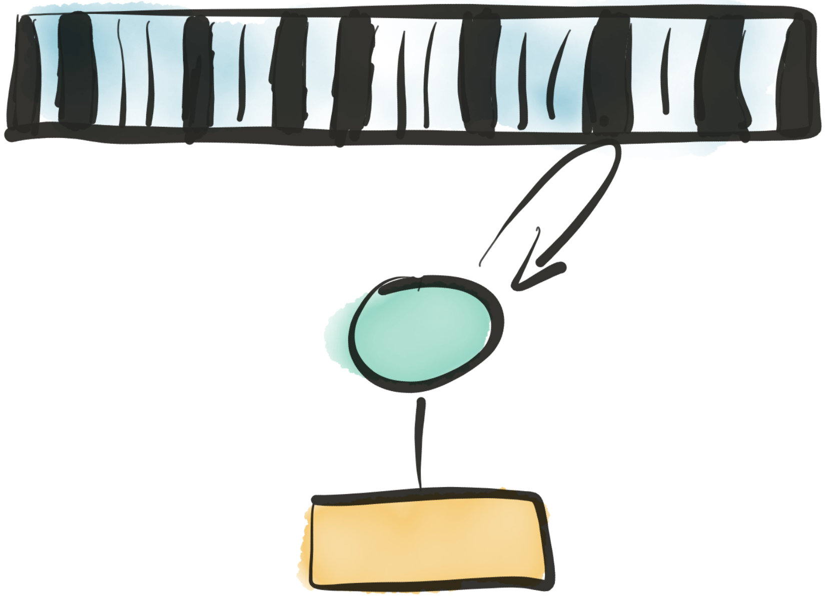
Do I need to look in this file to find the value for this key?

Size -> probability of false positive

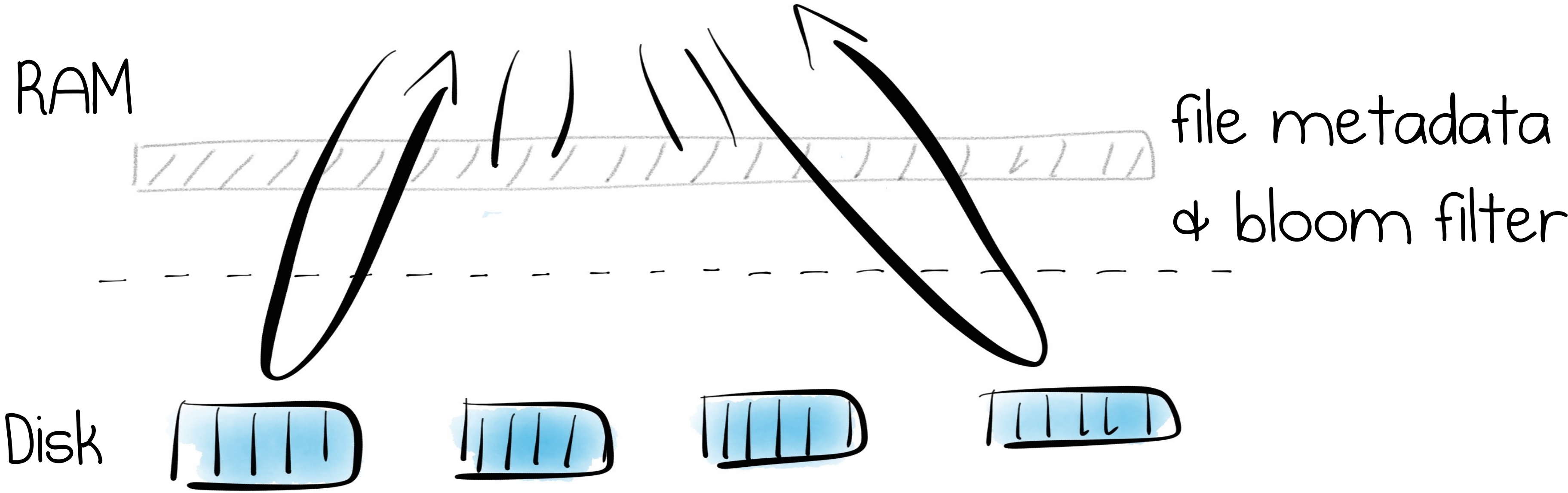


Bloom Filters

- Space efficient, probabilistic data structure
- As keyspace grows:
  - p(collision) increases
  - Index size is fixed



Many more degrees of freedom for optimising reads



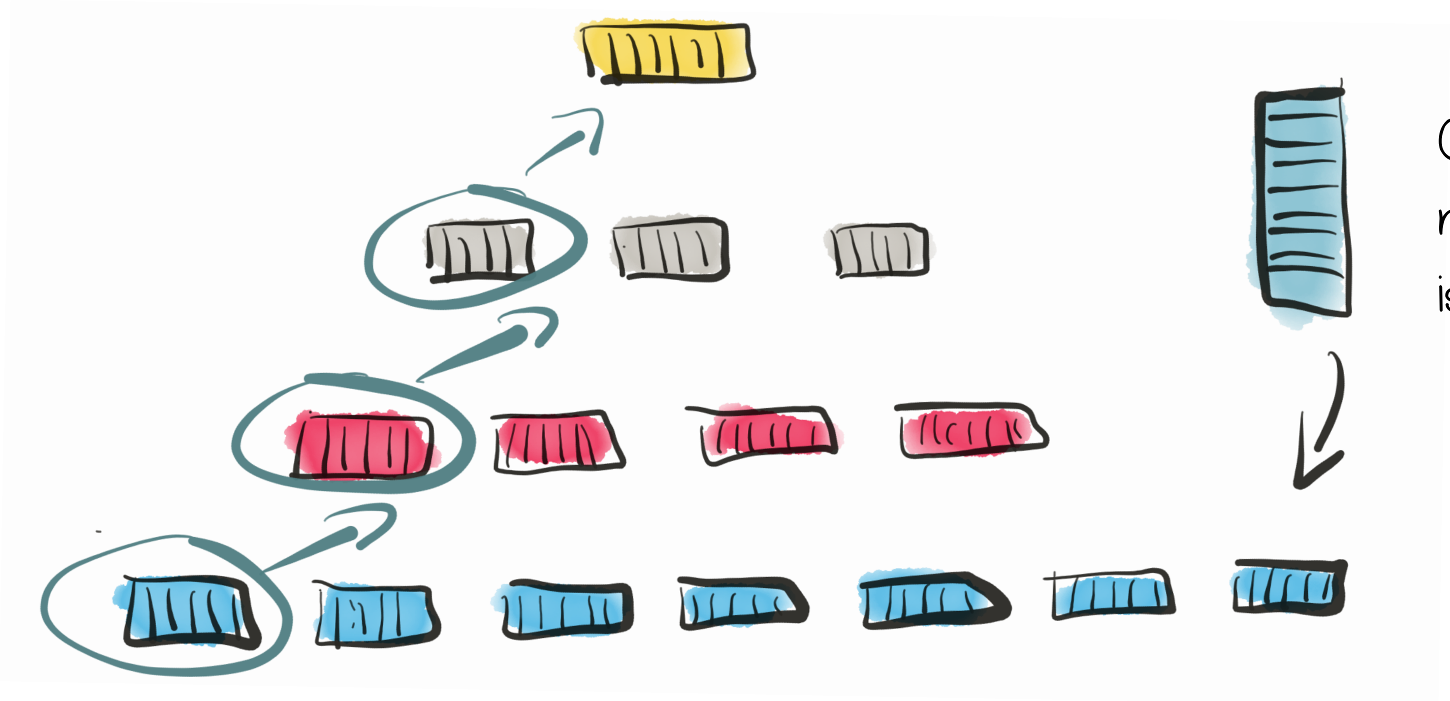
Log Structured Merge Trees

- A collection of small, immutable indexes
- All sequential operations, de-duplicate by merging files
- Index/Bloom in RAM to increase read performance

Subtleties

- Writes are 1 x IO (blind writes), rather than 2 x IO's (read + modify)
- Batching writes decreases write amplification. In trees leaf pages must be updated.

Immutability => Simpler locking semantics

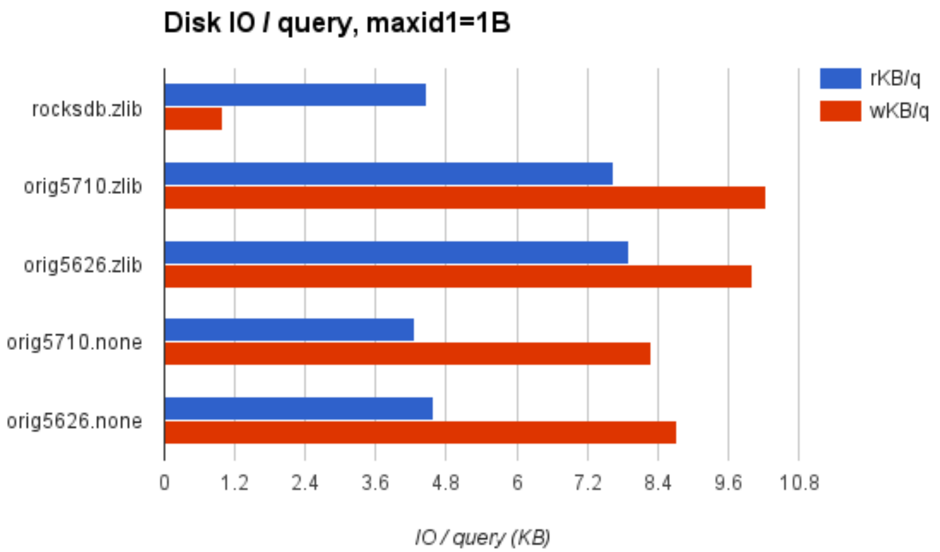


Only
memtable
is mutable

Does it work?

Lots of real world examples

Measureable in the real world



- Innodb vs MyRocks results, taken from Mark Callaghan's blog: http://bit.ly/2mhWT7p
- There are many subtleties. Take all benchmarks with a pinch of salt.

confluent

Elements of Beauty

- Reframing the problem to be Log-Centric. To go with the grain of the system.
- Optimise for the harder problem
- Compartmentalises writes (coordination) to a single point. Reads -> immutable structures.

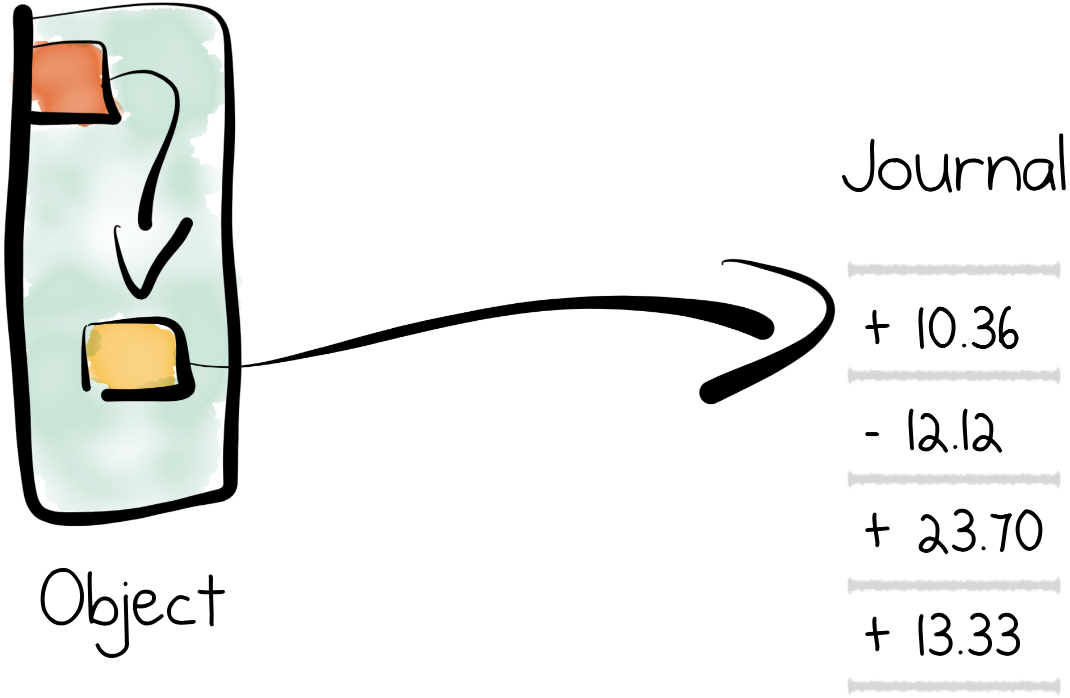
Applies in many other areas

- Sequentiality
  - Databases: write ahead logs
  - Columnar databases: Merge Joins
  - Kafka
- Immutability
  - Snapshot isolation over explicit locking.
  - Replication (state machines replication)

Log-Centric Approaches Work in Applications too

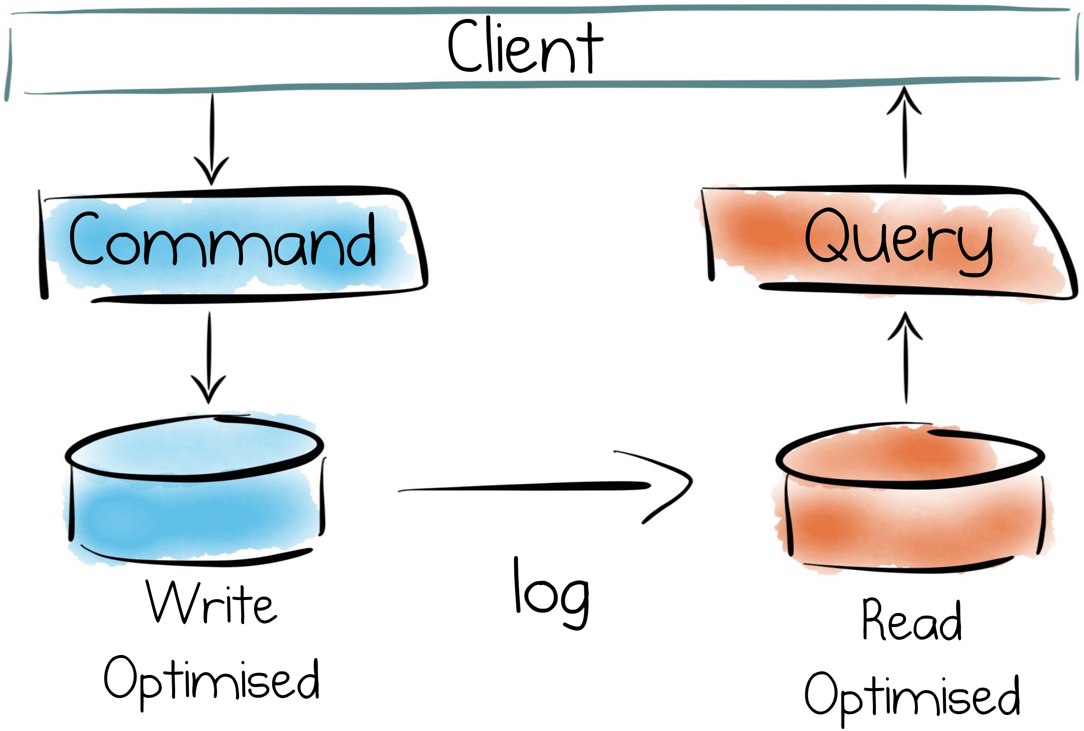
Event Sourcing

- Journaling of state changes
- No "update in place"



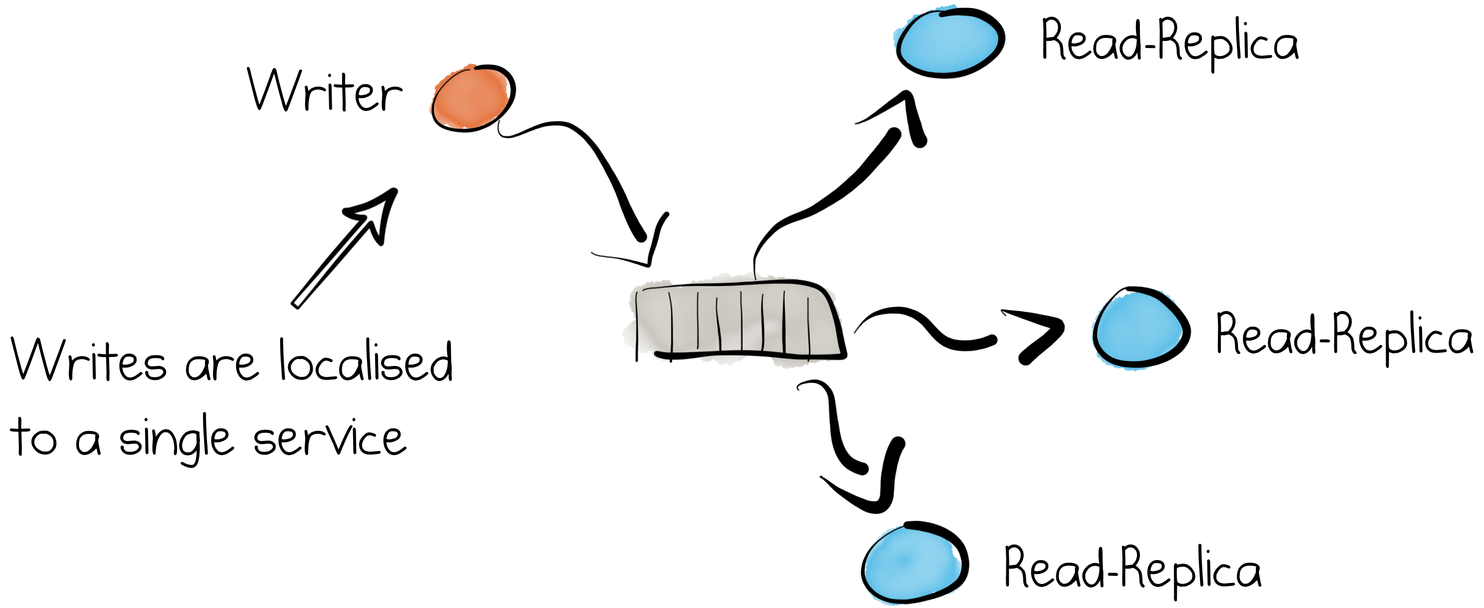
confluent

CQRS

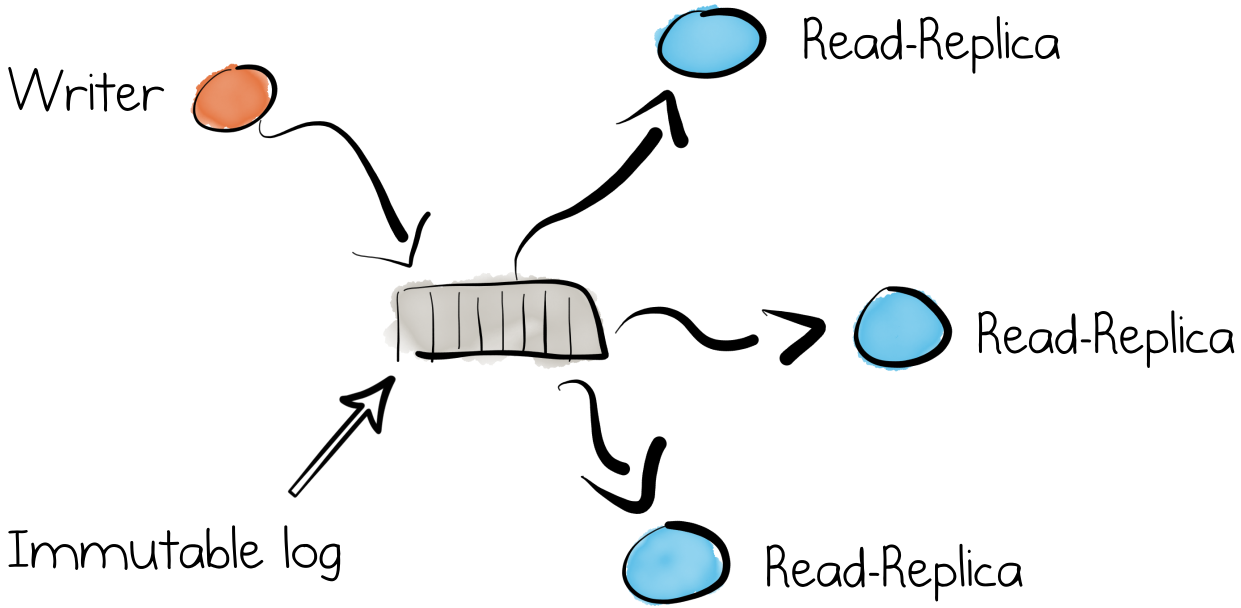


How Applications or Services share state

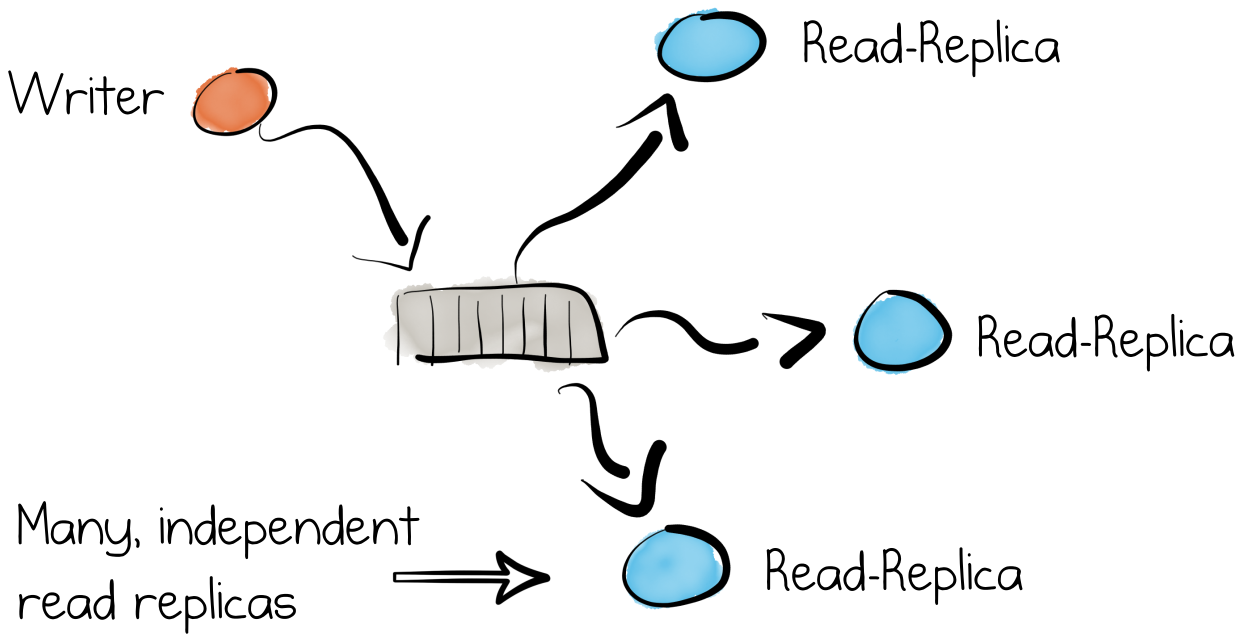
Log-Centric Services



Log-Centric Services



Log-Centric Services



confluent

Elements of Beauty

- Reframing the problem to be Log-Centric. To go with the grain of the system.
- Optimise for the harder problem
- Compartmentalises writes (coordination) to a single point. Reads -> immutable structures.

Decentralised Design

In both database design as well as in application development

The Log is the central building block

Pushes us towards the natural grain of the system

The Log

A single unifying abstraction

References

LSM:

- benstopford.com/2015/02/14/log-structured-merge-trees/
- smalldatum.blogspot.co.uk/2017/02/using-modern-sysbench-to-compare.html
- www.quora.com/How-does-the-Log-Structured-Merge-Tree-work
- bLSM paper: http://bit.ly/2mT7Vje

Other

- Pat Helland (Immutability) cidrdb.org/cidr2015/Papers/CIDR15_Paper16.pdf
- Peter Ballis (Coordination Avoidance): http://bit.ly/2m7XxnI
- Jay Kreps: I Heart Logs (O'Reilly 2014)
- The Data Dichotomy: http://bit.ly/2hk9c2K

Thank you

@benstopford

http://benstopford.com

ben@confluent.io

confluent