Plunge into FoundationDB documentation

for 2018-05-16 Helsinki Distsys Meetup group

By: Antti Vähäkotamäki - https://github.com/amv

PS. You can open https://foundationdb.org/ yourself!

Let’s start with something familiar: LevelDB

- Stores key-value pairs in a log-structured merge-tree (“one append-only file”)
- API: db.put(), db.get(), db.del(), db.batch(), db.write(), db.createReadStream()
- You can create a secondary index by atomically batching multiple writes:

```
db.batch()
        .put( ‘user:1234’,’{ “name”: “John Smith”}’ )
        .put( ‘index:John Smith’, ‘user:1234’ ).write();

db.createReadStream({ ‘gte’ : ‘index:John’, ‘lt’: ‘index:Joho’})
        .on( ‘data’, d => { console.log( d.key, ‘->‘, d.value ) } ); // All the Johns!
```

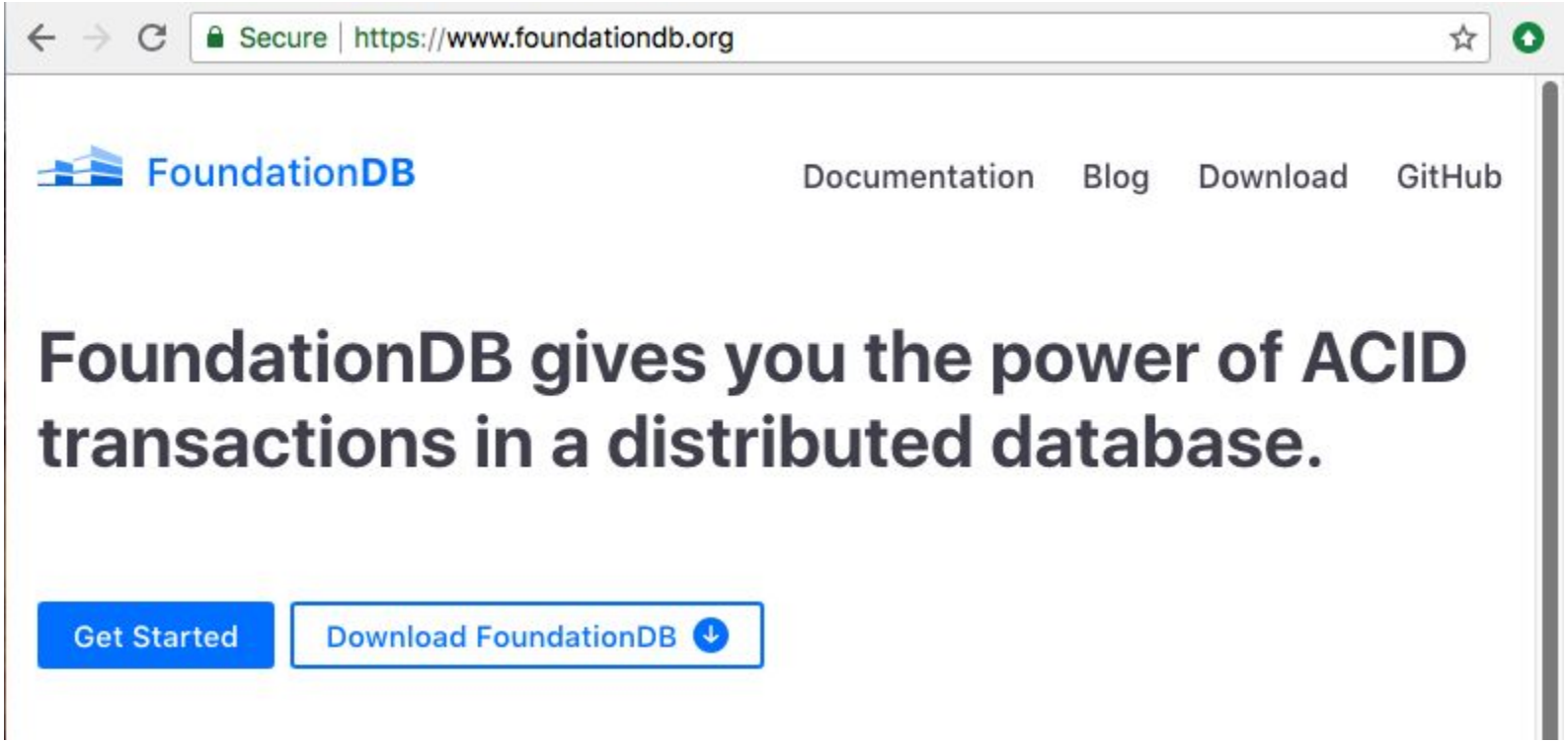
- The building blocks to solve most of your data storage needs?

Dear LevelDB, meet the real (distributed) world

- Storing data only on a **single machine** is prone to **unavailability** and **data loss**, so you would preferably write it to a LevelDB on multiple machines..
- .. and in case your **data grows** some day, you want to be able to **shard** it..
- .. and suddenly getting **atomicity, consistency and isolation** for your data operations became somewhat of a tall order :(
- Preferably coders would work with **transaction semantics**, so you just set up a set of transaction coordinators which communicate together and make transaction decisions, in addition to automatic repair and rebalancing operations based on a quorum.. **Simple & Easy**... right... ?

(this is why most distributed DB products give you “local transactions” that span only a single entity)

FoundationDB: Well they pretty much did just that.



Multi-model data store. FoundationDB is multi-model, meaning you can store many types data in a single database. All data is safely stored, distributed, and replicated in the Key-Value Store component.

Easily scalable and fault tolerant. FoundationDB is easy to install, grow, and manage. It has a distributed architecture that gracefully scales out, and handles faults while acting like a single ACID database.

Industry-leading performance. FoundationDB provides amazing performance on commodity hardware, allowing you to support very heavy loads at low cost.

Ready for production. FoundationDB has been running in production for years and been hardened with lessons learned. Backing FoundationDB up is an unmatched testing system based on a deterministic simulation engine.

Open source. We encourage your participation in our open source community! Join us in technical and user discussions on the community forums, and learn how to contribute.

Site Map

The full contents of this documentation site are listed below. If you are having trouble finding something in particular, try the search box in the navbar.

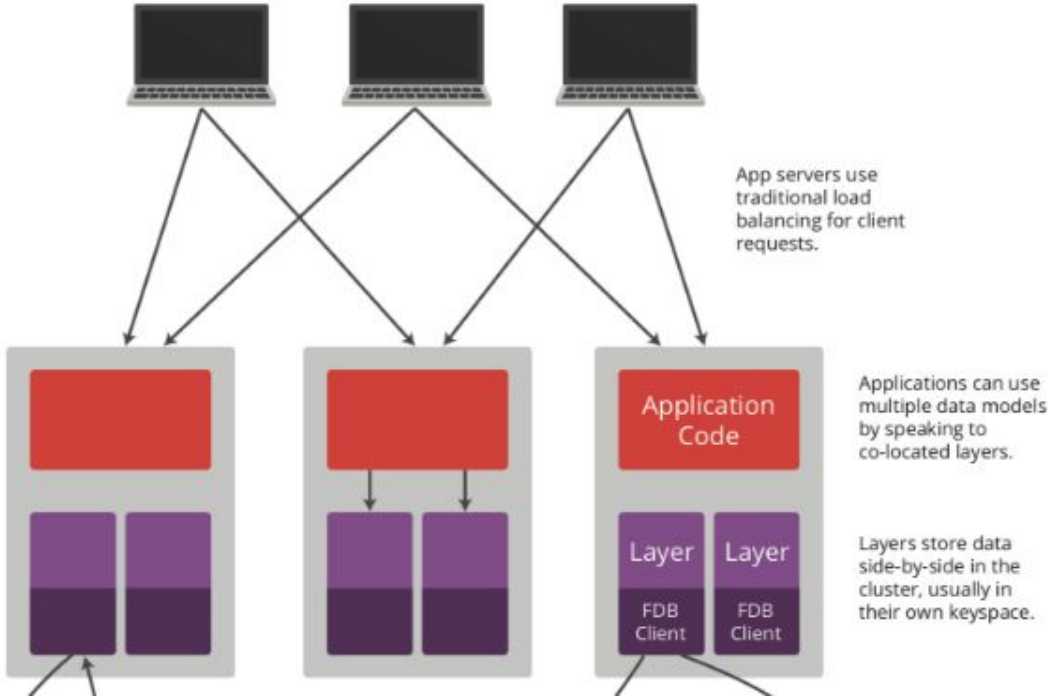
- Layer Concept
  - The old way
  - The FoundationDB way
- Anti-Features
  - What is an anti-feature?
  - Data models
  - Query languages
  - Analytic frameworks
  - Disconnected operation
  - Long-running read/write transactions

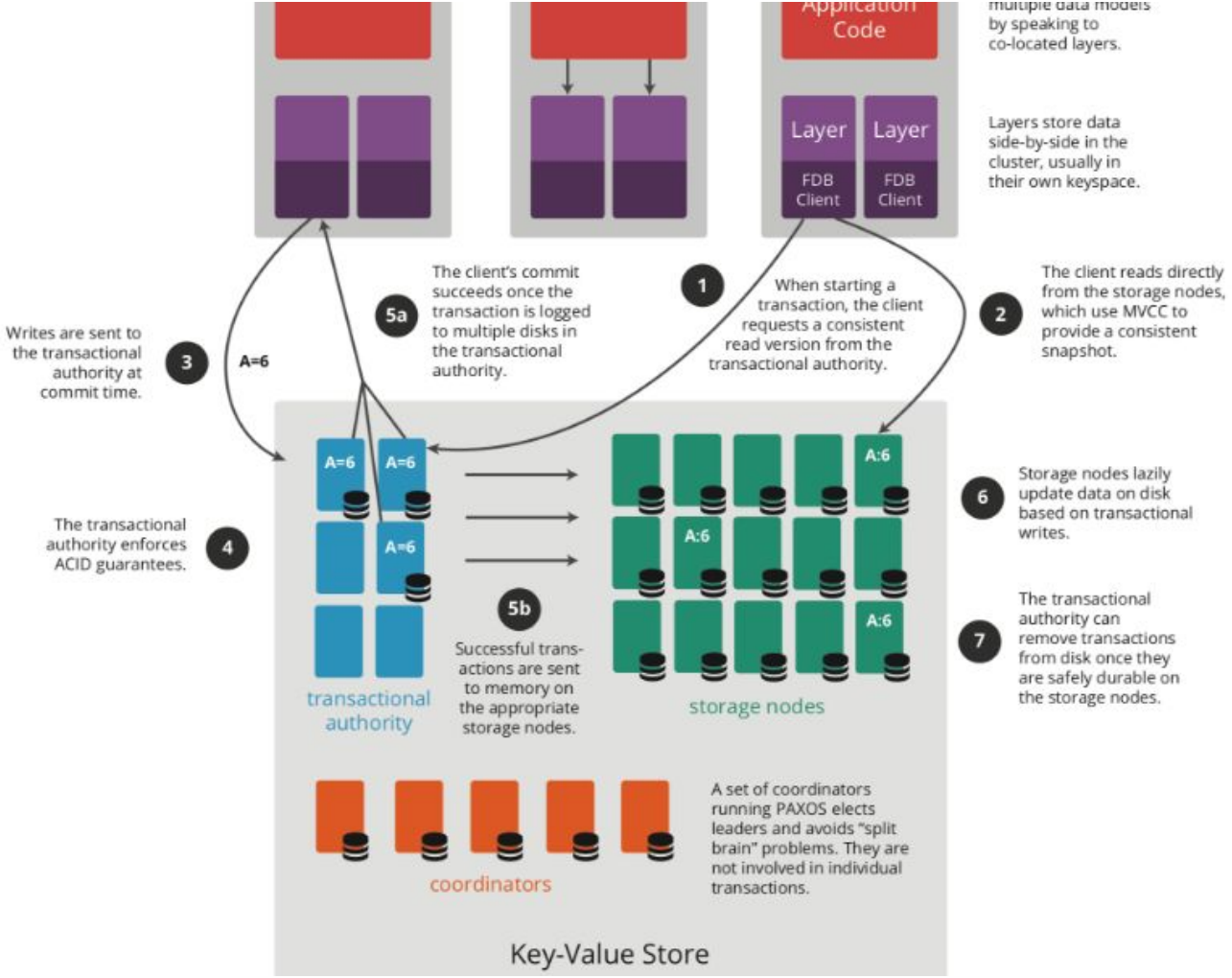
Background info in the “Why FoundationDB”-section

- Transaction Manifesto: Full ACID transactions are a powerful tool with significant advantages. To develop applications using modern distributed databases, transactions are simply essential.
- The CAP Theorem has been widely discussed in connection with NoSQL databases and is often invoked to justify weak models of data consistency. The truth is that the CAP theorem is widely misunderstood based on imprecise descriptions and actually constrains system design far less than is usually supposed.
- Consistency is fundamental to the design of distributed databases. However, the term *consistency* is used with two different meanings in discussions of the ACID properties and the CAP theorem, and it's easy to confuse them.
- Scalability: FoundationDB is horizontally scalable, achieving high performance per node in configurations from a single machine to a large cluster. We also offer elasticity, allowing machines to be provisioned or deprovisioned in a running cluster with automated load balancing and data distribution.

Key-Value Store Logical Architecture

DISCLAIMER: *Please do not try to infer system properties from this diagram.*
For that information, please see the Key-Value Store Features and
Known Limitations, or ask us a question.



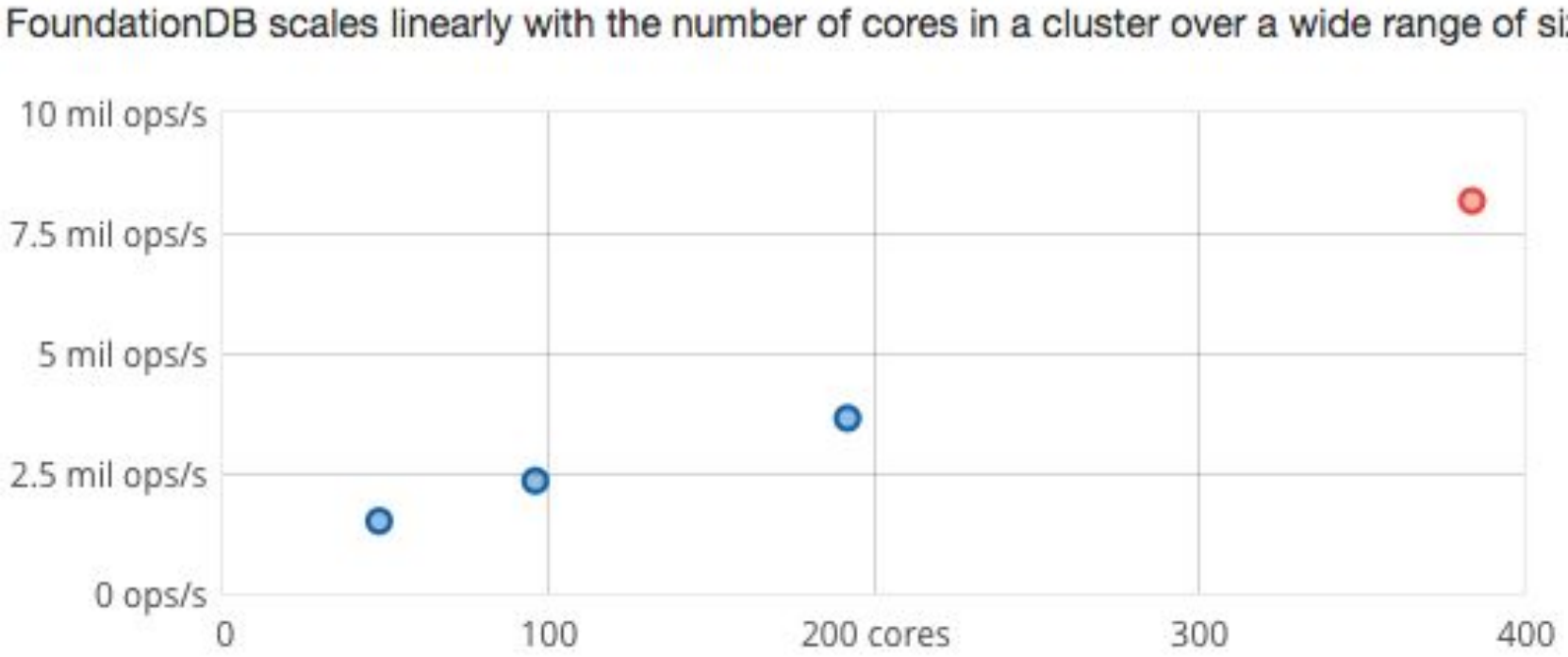

Application Code
Layer
Layer
FDB Client
FDB Client
multiple data models by speaking to co-located layers.
Layers store data side-by-side in the cluster, usually in their own keyspace.
1
When starting a transaction, the client requests a consistent read version from the transactional authority.
2
The client reads directly from the storage nodes, which use MVCC to provide a consistent snapshot.
3
A=6
Writes are sent to the transactional authority at commit time.
5a
The client's commit succeeds once the transaction is logged to multiple disks in the transactional authority.
A=6
A=6
A=6
transactional authority
4
The transactional authority enforces ACID guarantees.
5b
Successful transactions are sent to memory on the appropriate storage nodes.
A:6
A:6
A:6
storage nodes
6
Storage nodes lazily update data on disk based on transactional writes.
7
The transactional authority can remove transactions from disk once they are safely durable on the storage nodes.
coordinators
A set of coordinators running PAXOS elects leaders and avoids "split brain" problems. They are not involved in individual transactions.
Key-Value Store

- The Foundation
  - Scalable
  - ACID transactions
  - Fault tolerance
  - Replicated Storage
  - Ordered Key-Value API
  - Watches
  - Atomic Operations
  - OLTP and OLAP

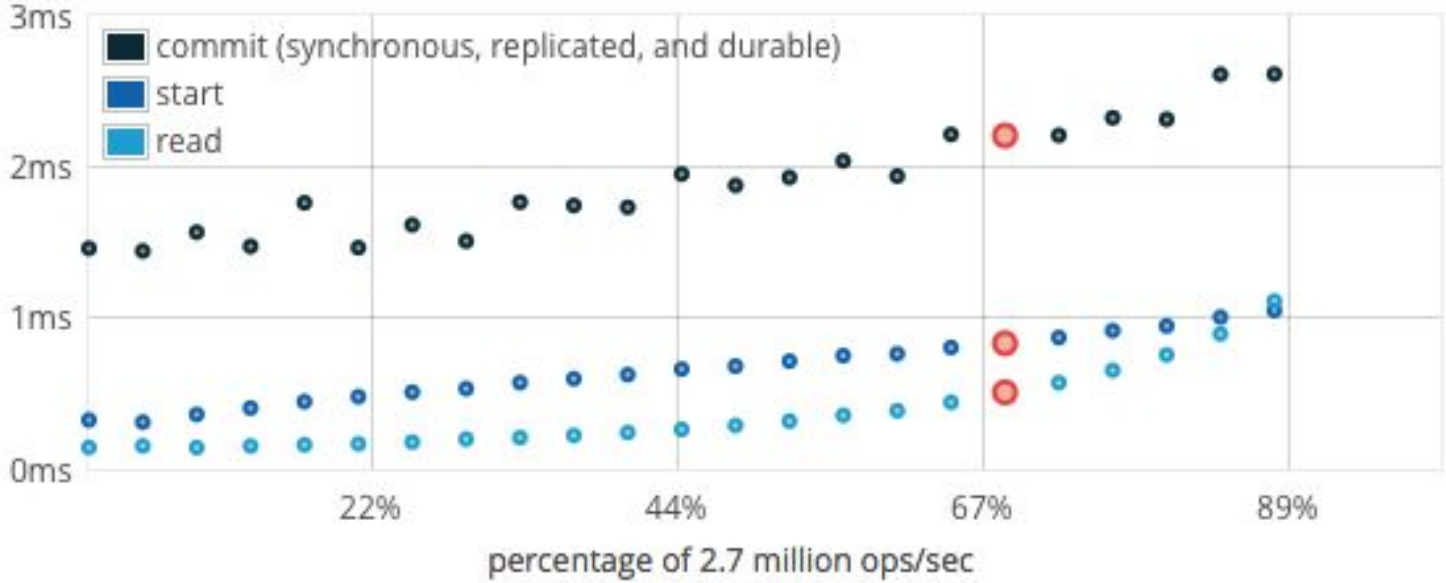
- Performance
  - Low, predictable latencies
  - Load balancing
  - Bursting
  - Distributed Caching
- Concurrency
  - Non-blocking
  - Concurrent Connections
  - Interactive Transactions

- Operations
  - Elastic
  - Datacenter Failover
  - Self Tuning
  - Deploy Anywhere
  - Backup

Some fancy performance benchmark images



FoundationDB has low latencies over a broad range of workloads that only increase modestly as the cluster approaches saturation.



The latency graph uses a 12-machine cluster in which each machine has a 4-core (E3-1240) processor and a single SATA SSD. We ran a FoundationDB server process on each core, yielding a 48-process cluster.

- Fault Tolerance
  - What is fault tolerance?
  - Distributed and replicated
  - Data distribution strategy
  - Independence assumptions
  - Other types of failure

- Known Limitations
  - Design limitations
    - Large transactions
    - Large keys and values
    - Spinning HDDs
    - Key selectors with large offsets are slow
    - Not a security boundary
  - Current limitations
    - Long running transactions
    - Cluster size
    - Database size
    - Limited read load balancing

Also one qualm comes up when Googling...

The memory required to run the **local development server:**

4 Gb

:D

Design Recipes

Learn how to build new data models, indexes, and more on top of the FoundationDB API. For more background, check out the Client Design documentation.

- Blob: Store binary large objects (blobs) in the database.
- Hierarchical Documents: Create a representation for hierarchical documents.
- Multimaps: Create a multimap data structure with multiset values.
- Priority Queues: Create a data structure for priority queues supporting operations for push, pop_min, peek_min, pop_max, and peek_max.
- Queues: Create a queue data structure that supports FIFO operations.
- Segmented Range Reads: Perform range reads in calibrated batches.
- Simple Indexes: Add (one or more) indexes to allow efficient retrieval of data in multiple ways.
- Spatial Indexing: Create a spatial index for the database.
- Subspace Indirection: Employ subspace indirection to manage bulk inserts or similar long-running operations.
- Tables: Create a table data structure suitable for sparse data.
- Vector: Create a vector data structure.

A simple transactional function to store a single blob with this strategy would look like:

```
CHUNK_SIZE = 10000

blob_subspace = fdb.Subspace(('myblob',))

@fdb.transactional
def write_blob(tr, blob_data):
    length = len(blob_data)
    if not length: return
    chunks = [(n, n+CHUNK_SIZE) for n in range(0, length, CHUNK_SIZE)]
    for start, end in chunks:
        tr[blob_subspace[start]] = blob_data[start:end]
```

The blob can then be efficiently read with a single range read:

```
@fdb.transactional
def read_blob(tr):
    blob_data = ''
    for k, v in tr[blob_subspace.range()]:
        blob_data += v
    return blob_data
```

So… What if network breaks after commit success?

```
# Increases account balance and stores a record of the deposit with a unique depositId
@fdb.transactional
def deposit(tr, acctId, depositId, amount):

    # If the deposit record exists, the deposit already succeeded, and we can quit
    depositKey = fdb.tuple.pack(('account', acctId, depositId))
    if tr[depositKey].present(): return

    amount = struct.pack('<i', amount)
    tr[depositKey] = amount

    # The above check ensures that the balance update is executed only once
    balanceKey = fdb.tuple.pack(('account', acctId))
    tr.add(balanceKey, amount)
```

Open Source Layer Microservice Docker Images?

- Haven’t seen them yet - was fully open sourced by Apple less than a month ago… but sounds like a pretty good idea if a good monitoring standard emerges, right?
- Before Apple bought it, even a SQL capable layer was introduced, but I couldn’t find what the status or performance is today. I saw some past benchmarks showing that it performed with ~50% of the speed of MySQL, which would IMHO be pretty impressive if true for normal use cases.

Thanks!

Source of pretty much everything:

https://www.foundationdb.org/