Read and write considered harmful

ACCU Bristol, April 2018

Hubert Matthews
hubert@oxyware.com
Why this talk?
Overview

1) Basics of Read/Write
2) Business processes, rules and schemas
3) Performance, scaling, concurrency
4) Six questions about data
5) Asynchrony and queues
6) Structure changing
7) State management
Reads

- Can be cached, at multiple levels (e.g. CPU)
- Caching is transparent (mostly)
- Idempotent (can be retried without side-effects)
- Can be partitioned easily (routing)
- Access control rules only
- Synchronous and blocking
- Scalable bandwidth – fanout reduces contention
Writes

• Caching is horrible for writes
  – writeback/through, eviction policy, coherence, etc
• Scaling writes is horrible – fan-in creates contention
• Sharding works well only for primary key writes
• Access control rules plus update rules
• Can be delayed or asynchronous
• Idempotence is a design issue/choice
Dependencies

- Even simple code has dependencies caused by read and write
- Asymmetric – caller object doesn’t know who calls it
- Makes testing difficult
  - Substitution
  -Mocks, etc
- Introduces notion of push and pull
REST APIs and rules

REST = getters/setters on steroids

- Industrial-scale anti-pattern
- Separates code and data
- Opposite of encapsulation
- Very non-OO
- Duplicated logic/rules in every client
- Example: stock_level >= 0
- If rule is broken (stock_level == -1) where is the bug?
REST APIs and schemas

Tradeoffs: early/late validation failure, schema migration, versioning, code/rule duplication
REST APIs and processes

REST = CRUD
- Statechart has one state and four transitions
- OK for metadata

Real processes have multiple states
- Have different entities per state (possibly sub-entities)
Typical system scaling path

- Application is too slow
- Get more front-end boxes (scale R+W)

- Application is still too slow
- Get bigger DB box (scale R+W)

- Run out of read bandwidth
- Replicate or cache data (scale R)

- Run out of write bandwidth
- Shard/partition data on primary key (scale R+W)

- Create separate services per entity/component
- Cross-service joins done in client (scale entities)

stop when it's fast enough
Scaling problems

• Partitioning or sharding works to an extent
  – If access is strongly biased around primary key

• Nasty to scale cross-partition operations
  – Particularly for write (usually not idempotent)
  – Partial failure on write, cross-box transactions, concurrency, latency, etc (classic Waldo paper)
  – Service boundaries aligned with operation boundaries and failure boundaries

• Bulk access to multiple records can cause N+1 access problem (get primary keys then N single-row accesses – beware of REST and ORMs)
Avoid sharing mutable data

- Shared mutable data is the evil of all computing!
- Read-only data can be shared safely without locks
- Const is your friend
- Pure message-passing approach avoids this
Shared writes don't scale

Operation Scalability
(on 4 processors x 4 cores AMD machine)

Throughput, op/sec

Thread Count

Read private  Read shared  Write private  Write shared  RMW private  RMW shared

(graphic by Dmitry Vykov, http://www.1024cores.net, CC BY-NC-SA 3.0)
Why shared writes don't scale

- Caches have to communicate to ensure coherent view
- MESI protocol passes messages between caches
- Shared writes limited by MESI comms

![Diagram showing cache levels and MESI protocol](image-url)
6 questions about data access

<table>
<thead>
<tr>
<th>PK</th>
<th>hashing, partitioning, op==</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-PK</td>
<td>search, secondary indexes, full-text search</td>
</tr>
<tr>
<td>Range scans</td>
<td>ordering, iteration, bulk vs. single values, op&lt;</td>
</tr>
<tr>
<td>R/W ratio</td>
<td>caching, cost of lookups vs. cost of updates</td>
</tr>
<tr>
<td>Working set</td>
<td>how big is the commonly accessed set of data (RAM)</td>
</tr>
<tr>
<td>Consistency</td>
<td>exact results vs. fast approximations, eventual consistency, replication, batched updates</td>
</tr>
</tbody>
</table>

Strongly related to the operational profile
1. Primary key access

• Most common form of access
  – Database primary key
  – std::map/unordered_map

• Can use hashing \([O(1)]\), binary search \([O(\log N)]\) or linear search for small \(N\) \([O(N)]\)

• Requires only operator==

• Partition on primary key into multiple parts that can operate in parallel or to avoid contention

• Examples: product catalogue, customer records, sticky web sessions, NoSQL, memcache, REST
2. Non-primary key access

- Finding items by value, not by key
- Need for secondary indexes (e.g. database indexes)
- Search on parts of a record
- Metadata search (date/time, etc)
- Full-text search
- May require substantially more work to build compared to PK-based access
- Usually slower than PK access for lookup
3. Range scans and sequential access

- Requires ordering, i.e. \texttt{operator<}, (ordering costs)
- Requires iterators/cursors/traversal state
- Seek then scan – first find is slow, then fast
- Dense linear access and prefetch
- Watch out for read/write amplification
- Bulk, not single record, access – may require bulk aggregate operations rather than \(N\) times single record operations for speed (DB \(N+1\) problem)
## 4. Read/write ratio

<table>
<thead>
<tr>
<th>High reads</th>
<th>High writes</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Caches are effective</td>
<td></td>
</tr>
<tr>
<td>• Cache writethrough/back</td>
<td></td>
</tr>
<tr>
<td>• Cache eviction policy</td>
<td></td>
</tr>
<tr>
<td>• Cache coherency</td>
<td></td>
</tr>
<tr>
<td>• Index structures useful</td>
<td>• Caches don't help much (except for metadata)</td>
</tr>
<tr>
<td></td>
<td>• Locking overhead</td>
</tr>
<tr>
<td></td>
<td>• Index structures require updating</td>
</tr>
</tbody>
</table>

Not all data in a system has similar R/W ratios e.g. metadata is often read-heavy
5. Working set size and skew

• How much of the common data will fit in main memory, the L1/L2/L3 cache
• Will the index structures fit but not the main data
  – Index data tends to be “hot”, main data may be “cold”
• Depends on data access patterns – 80/20 rule
6. Consistency

• Do all copies of the data need to be exactly up-to-date right now
  – ACID, 2-phase commit, centralised, locking, slow
• How often are copies updated
• Batch updates (e.g. overnight)
• Data vs. metadata (transactions vs. reference data)
• ACID vs. BASE (eventual consistency)
  – BASE allows for decoupled asynchronous systems
Notes
1. Arrows show connections; they do not indicate data flow.
2. Does not take into account the application usage patterns that impact scaling.
3. Does not show staging servers.
4. Letters A-G refer to descriptions of servers in ATG Multiple Application Integration Guide.

Key
- □: Internal users
- □: External users
- ⬤: Database
- ➤: ODBC/Native connection
- ➡: JDBC connections
- FW: Potential firewall
- ATG DW: Data warehouse database
Read or write, pull or push?

- Is X reading Y or writing to it?
- Or both at different times?
- Is X pushing or Y pulling?
- Where is the thread of control?
- Is this a full batch update or a partial incremental change?
- Is this an asynchronous push (fire-and-forget) or a synchronous blocking call?

Diagrams like this are often useless.
**Full vs incremental**

- Full changes allow the state to be reset on a regular basis
  - Prevents build-up of errors or divergence from base data
  - Slow, long latency, partial failure problems
  - One big transaction

- Incremental changes are fast but don’t guarantee to keep state changes synchronised
  - Lost messages because of unavailability
  - Transactionality only on each update
**Lambda Architecture**

- **Batch Layer**
  - Immutable master data
  - Batch recompute
  - Precompute views
  - Batch layer does full updates from read-only data

- **Speed Layer**
  - New data stream
  - Process stream
  - Real-time increment
  - Real-time views
  - Increment views
  - Speed layer does incremental real-time updates

- **Serving Layer**
  - Lambda architecture
  - View 1, View 2, View N
  - Batch views
  - Merge
  - Merge views
  - Query
Reader/writer vs data flow

- Readers and writers view hides inherent data flow in systems
- Split R and W into microservices
  - Separates rules, performance, scaling, access control, etc
  - Often W and R are very different
  - W usually reads from somewhere
Sync vs async systems

• ACID is hard to scale, partition, get right, can promote failures, makes for a more fragile system as everything has to be up all the time (brittle)
  – 10 sync systems with 99% uptime => 90% uptime
  – 10 async systems => 99% uptime for end system

• Can maybe delay writes or cover them up (lambda arch)

• Reads are sync but recent data may be sufficient, particularly for metadata (caching helps lots)
Data flow and sync/async

- Data flows can be point-to-point or broadcast
- Can be synchronous or asynchronous
  - Message queues provide simple sync intermediary
  - Flat file batch transfer is popular for a reason
- Queues can also be event stores with reread
  - c.f. Kafka => LinkedIn
  - Makes queue reads idempotent
**Content management example**

- **Serve**
- **Edit**
- **Store**

CMS

- Editing (complex, infrequent) and serving (high performance, read-only, secure) are intertwined.

- Editing is not exposed, complexity matches domain.

- Serving is simple, isolated, fast and secure (e.g. CDN).

- Data flow approach keeps two APIs separate:
  - Security, clarity of purpose
  - Separation of concerns
  - Horizontal not vertical thinking
• Data flow makes you think about where data comes from and goes to
  – “Who is actually going to read the data I’m writing?”
• Microservices may have two APIs for sending and receiving
  – “Vertical” thinking may lead to trying to fit both into one API
Command Query Representation Separation (CQRS)

- Need to change the structure from the form that best suits the write API to the structure that best suits the read API
- Can be synchronous or asynchronous transformation
CQRS examples

- Structure change can be on read or on write
  - Twitter does it on read for high-value users, not write
- Log-structured merge systems do this internally and asynchronously (e.g. HBase, RocksDB)
- Other examples
  - Time-series databases
  - Event sourcing
  - Struct-of-arrays vs array-of-structs (e.g. non-OO, data oriented)
  - Columnar analytics databases (e.g. Redshift, BigQuery)
State management

- Read and write focus doesn’t help manage state across a complex system
- State management needs to address:
  - Transactions vs eventual consistency
  - Failure management
  - Availability and MTTR
  - Immutability
- Align txn boundaries with failure and aggregate boundaries
  - REST API: /resourceA/1/resourceB/2
  - Fragmentation and transactionality problems
Failure management

• Distributed systems can suffer from partial failures on writes
  – Writes in distributed are inherently concurrent
  – Recovery and resynchronising state is “fun”

• Idempotent writes allow for replay and deduping
  – Make deduping easy: serial number, timestamp, etc
  – Repeatable queues are useful (e.g. Kafka, flat files)

• Checkpointing of known good state
  – Point-in-time recovery
  – Full vs incremental update problem again
Bell-LaPadula and Biba models

**Bell-LaPadula**: confidentiality

**Biba**: integrity

**diametrical opposites**
Immutability

• Functional programming languages have immutable data
  – Make sharing and reasoning about data easier
• Russian Doll caching, MVCC
  – Change the key not the value
• Pets vs cattle – infrastructure
• SSA – compilers and CPU reservation stations
• Lambda architecture – immutable master data

Immutability makes things simpler
Availability

Availability = \frac{MTBF}{MTBF + MTTR}

(where MTBF = mean time between failures
MTTR = mean time to repair)

• Maximise MTBF by “normal” means:
  – Good software practices, hardware failover, reliable well-known technology choices

• Minimise MTTR by making systems easier to understand, debug and restart
  – Minimise state management (txn redo logs, fsck, etc)
  – Use immutability where possible
Read and write are too low level

- They don’t help you to design or analyse systems
  - They are the assembler-level of data (CRUD)
- They don’t relate to the larger picture
- It is too easy to deal with them in isolation
  - REST APIs are an all-too common example
- Looking at data flows, push vs. pull, sync/async, business processes, operational profiles, state management, etc are much more fruitful approaches