Data consistency in 3D

(It’s the invariants, stupid)

Marc Shapiro
Masoud Saieda Ardekani
Gustavo Petri

This talk is about...

Understanding consistency
• Primitive consistency mechanisms
• How primitives compose models
• How models relate / differ
• What they cost
Understanding invariants
• Some interesting classes of invariants
Relating consistency to invariants
• Which primitives guarantee which invariants
Useful intuitions for app. and system designers

Shared database

q.push(e)  c.inc()
q.val()  c.val()

Geo-replicated database

q.push(e)  c.inc()
q.val()  c.val()
Consistency

More replicas:
- Better read availability, responsiveness, performance, etc.
- More work to keep replicas in sync

Consistent = behavior similar to sequential:
- Satisfies specs: does $q$ behave like a queue?
- Replicas agree: is $q$ identical everywhere?
- Objects agree: is $|q| \leq c$?
- Same flow of time? $q1.push()$ before $q2.push()$

Consistency opportunities and costs

CAP
Availability
⇒ Parallelism keeps the hardware busy
⇒ More implem. options, scalable

But consistency constrains order of events:
- Delay delivery
- Stale reads
- Waits, synchronisation (mutual wait)

Keeping track of order requires metadata
Significant!

Costs illustrated

Termination Latency of Update Transactions (ms)

Credit: Masoud Saeida Ardekani

Strict Serialisability

Invariant

Invariant

Invariant
Eventual consistency

Strong vs. weak?

Predictable
Strict Serialisability
snapshot Isolation
PRAM
Eventual Consistency
Low performance
High performance
Hard to program

Predictable
Strict Serialisability
Snapshot Isolation
PRAM
Eventual Consistency
Low performance
High performance
Hard to program

Strong vs. weak?

Predictable
Strict Serialisability
Snapshot Isolation
PRAM
Eventual Consistency
Low performance
High performance
Hard to program

Predictable
Strict Serialisability
Snapshot Isolation
PRAM
Eventual Consistency
Low performance
High performance
Hard to program

Figure 4-1: A partial order to relate various isolation levels.

For all intermediate levels, we have also developed corresponding guarantees that can be provided to transactions as they execute. As in the previous chapter, the levels defined for running transactions are similar to the corresponding levels for committed transactions.

The rest of this chapter is organized as follows. In Section 4.1, we present our specifications for PL-2+. In Section 4.2, we present definitions for PL-2L. In Section 4.3, we describe specifications of Snapshot Isolation. We discuss a new isolation level called Forward Consistent View in 4.4 that has been inspired by Snapshot Isolation. We describe a level that captures the essence of Oracle’s Read Consistency in Section 4.5 and compare it with level PL-2L. Cursor Stability is presented in Section 4.6. Section 4.7 describes update serializability, a consistency guarantee that is useful for read-only transactions, and compares it with PL-2+ and serializability. Finally, in Section 4.8, we extend our definitions for intermediate levels to provide guarantees for executing transactions.

4.1 Isolation Level PL-2+

Isolation level PL-2+ is motivated by the fact that certain applications only need to observe a consistent state of the database and serializability may not be required, e.g., a read-only transaction in an inventory application may simply want to observe a consistent state of the current orders and in-stock items. It is the weakest level that ensures that integrity constraints are not observed as violated as long as update transactions modify the database consistently and are serializable.

Adya 1999

Transaction models

Eventual serializability
Three classes...

<table>
<thead>
<tr>
<th>Gen1</th>
<th>Object value</th>
<th>Total order of operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>PO</td>
<td>Relative ordering of operations</td>
<td>Visibility</td>
</tr>
<tr>
<td>EQ</td>
<td>State equivalence</td>
<td>Composition</td>
</tr>
</tbody>
</table>

Operation

generator: read, compute, generate effector
effactor: compute, write side-effect
Sequential execution:
- precondition $\implies$ invariant
- each effector individually safe

Sequential correctness

generator: read, compute, generate effector
effactor: compute, write side-effect
Sequential execution:
- precondition $\implies$ invariant
- each effector individually safe
Guarantee vs. semantics

Guarantee:
- Class of invariants that is always true
- Regardless of application code
- Assuming sequentially correct

Application can compensate for absence of guarantee
- e.g. Inv={ c ≥ 0 }, app: c.inc()

Data types

Register
- Update: assign with constant
  - Not commutative
  - Absorbing

High-level types
- Counter, ORset, Sequence: effectors commute
- Stock, Account, Queue: ¬ commute

Composed data
- + structural invariants

Replicated operation

u: state ↦ (retval, (state ↦ state))
Read one, write all (ROWA)
Deferred-update replication (DUR)

Sharded, geo-replicated

 arbitrary origin

sharded, parallel

 concurrent updates

~ read my writes
Type EQ invariants

- $A = B$
- $x.frienOf(y) \iff y.frienOf(x)$
- $x + y = \text{constant}$
- $\text{South} \cup \text{Boat} \cup \text{North} = \{\text{sheep, dog, wolf}\}$

Joint update to two objects

Atomicity (all-or-nothing) property of transactions

Protocol: single update message
- Asynchronous

Airplane reservation

- Allocate a seat to me
- Pay for the flight

Two EQ relations:
- $\text{paid} = \text{have_seat}$
- $\text{my} \; \text{money} + \text{airline} \; \text{money} = \text{constant}$

Ad-hoc grouping

(This txn also needs TO + snapshot)

**EQ/Composition axis**

Transaction groups operations
All-or-nothing effects:
- Deliver effectors indivisibly
  - packaged together
- $\approx 2$-phase commit

Snapshot reads:
- all generators read from
  - same set of effectors
- $\approx \text{trans. causal}$
- maintain versions
- $\approx \text{coordination}$

**EQ/Composition axis**

Transaction groups operations
All-or-nothing effects:
- Deliver effectors indivisibly
  - packaged together
- $\approx 2$-phase commit

Snapshot reads:
- all generators read from
  - same set of effectors
- $\approx \text{trans. causal}$
- maintain versions
- $\approx \text{coordination}$
**EQ/Composition axis**

Transaction groups operations

All-or-nothing effects:
- Deliver effectors indivisibly
  ‣ packaged together
- + same TOE
  ‣ \( \approx \) 2-phase commit

Snapshot reads:
- all generators read from same set of effectors
  ‣ maintain versions
- + same TO, VIS guarantees
  ‣ coordination

**Type PO invariants**

- \( \text{employee.manager.salary} \geq \text{employee.salary} \)
- \( S1; S2; S3 \equiv S1 \iff S2 \iff S3 \)
- \( \text{dog} \in S \iff \text{sheep} \in S \land \text{wolf} \in S \)
- Referential integrity
  "inode references disk block"
- \( \text{ACL} (u, p) \iff \text{access} (u, p) \)

Demarcation Protocol:
1. increase LHS by \( c \)
2. increase RHS by \( c' \leq c \)
\( \Rightarrow \) ordered delivery
No synchronisation: Available

**PO: transitive / causal visibility**

\( x = 100; y = 100 \)
\( \text{Inv} = \{ x \geq y \} \)

Ex 1:
- P1: \( x += 100 \)
- P2: if \( x > y \) then \( y += (x-y)/2 \)
- P3: \( x \geq y ? \)
- Transitive visibility \( \text{vis}^* \subseteq \text{vis} \)

Ex 2:
- P1: \( x += 100; d = 100 \)
- P2: if \( d > 0 \) then \( y += d/2 \)
- P3: \( x \geq y ? \)
- Causal visibility \( (\text{vis}; \text{po})^* \subseteq \text{vis} \)

\( \text{client is part of DB} \)
PO/Visibility axis

Visibility
- Which writes visible to reads

Transitive closure property
- Metadata
- System-wide

PO/Visibility axis

Monotonic client

- Read My Writes
- Monotonic Reads

Often assumed
- Buffer

Transitive, causal vis.
- Effector: metadata identifies set of predecessor effectors
- Delay delivery after predecessors
  - Read stale data
- Graph: unbounded
- Vector clock: $10^4 - 10^6$ entries $\times$ 8 bytes!
- Approximate VC: stronger order

Total/external causal

Total order extends causal order
Metadata: 1 single scalar
- but cost of total order
External: real-time clock
Gen1 invariants

\[ \text{Inv} = \{ x \geq 0 \} \]
\[ \text{u} = \{ x \coloneqq x - 1 \} \]
\[ \{ \text{Inv} \land 1 \leq x \} \text{ u } \{ \text{Inv} \} \]

Predict that \( \text{Inv} \) will be true after \( \text{u} \):
- Sequential: weakest precondition
- Generalises to bounded concurrency

Unbounded concurrency: no sufficient precondition
- Invariant is not stable
- Limit concurrency: escrow
- No concurrency: order updates

0 = unordered

Do replicas observe events in the same order?
Pick a unique number

Capricious TO effectors

Do replicas observe events in the same order?
Pick a unique number

Pick a number locally: capricious
gap: will arrive later?
- Non-monotonic: rollback
- Monotonic
  - Wait for gap to fill (Lamport 78)
  - Lost updates (LWW)
**Capricious TO Effectors**

- Do replicas observe events in the same order?
- Pick a unique number

- Gap: will arrive later?
  - Non-monotonic: rollback
  - Monotonic: wait for gap to fill (Lamport 78)
  - Lost updates (LWW)

- Capricious: Totem Pole

**Gapless TO Effectors**

- Do replicas observe events in the same order?
- Pick a unique number

- Gapless: no lost updates
- Consensus, 2PC to uniquely allocate next free number

**TO Generators**

- Do replicas observe events in the same order?
- Pick a unique number

- Separate from effectors
- Same order as effectors

**TO Effectors**

- Capricious: Totem Pole

### Three Dimensions

- Gen1 / Total Order
- PO / Visibility
- EQ / Composition
- SI
- LWW
- EC
- LAMPORT
- PSI
- NMSI
- SSER
- LIN
- SER
- CAP
Our system model (Section 2) is very general. The separation between generators and effectors allows for internal parallelism; if unusual, it reflects practical implementations [23]. Our total order axis (Section 3), classifies the degree of concurrency between operations to a single object, including only effectors or also generators, and accounts for both available (capricious) and consensus-based (gapless) approaches. The other two axes introduce mechanisms that relate multiple objects; however, they serve different purposes and have different costs. Visibility order (Section 4) relates reads to writes and involves maintaining a system-wide transitive closure, and aims to support PO-type invariants. Transactions (composition, Section 5) serves to enforce ad-hoc EQ and Gen*; a transaction is a one-operation grouping, requested by the application. In order to be intuitively useful, our classification simplifies the design space into three approximately linear axes (which we relate to application invariants). Obviously, this cannot account for the full complexity of the relations between models. We acknowledge the deficiencies of such a simplification. For instance, we flatten the visibility axis, and abusively assume that all TOG=TOE models must be gapless. We defend this simplification as practically relevant, even if not formally justified. We also ignored hybrid models, such as Update Serialisability [16]. We focus on client-monotonic models, as they are the most intuitive, and because monotonicity is trivial to implement. While the specifications of SER, NMSI, or RC do not require Monotonic visibility, all the actual implementations that we know of do provide it. Table 5 positions some major consistency models within the three axes. Compare for instance two prominent strong consistency models: SSER and LIN. While LIN considers single operations and single objects, SSER is a transactional model requiring All-or-Nothing and Snapshot. Also notice how the visibility axis differentiates SSER from SER, and NMSI from PSI. While our results are preliminary, we believe that this classification sheds light on the crowded space of distributed consistency guarantees, towards a better understanding of the application invariants enforced by each of them. We intend, in further work, to formalize our definitions and prove some interesting meta-properties. This work aims to be an step
Summary

Distributed, replicated data
- Improves read availability
- Parallel updates may violate invariants
- Guarantee: invariants maintained by system
  - System vs. application cost trade-off
    - Tools needed

3D consistency design space
- Total order (effectors, generators)
- Visibility order
- Transactional Composition

Work in progress

Creative Commons Attribution-ShareAlike 4.0 Intl. License

You are free to:
- Share — copy and redistribute the material in any medium or format
- Adapt — remix, transform, and build upon the material for any purpose, even commercially, under the following terms:

  Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

  ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

4 session guarantees

Monotonic reads
- Client / No rollback: r3 must include w1

Monotonic writes
- Global / No rollback: r3 must include w1

Read My Writes
- Client / RMW: r2 must include w1

Writes Follow Reads
- Global / WR dependence: w3 must follow w1