Principles of Software Design
Persistence and Databases

Robert Lukoťka
lukotka@dcs.fmph.uniba.sk
www.dcs.fmph.uniba.sk/~lukotka

M-255

How to save the state of your application?

90's - It was quite easy. User pressed the save button. No shared state, single threaded, no need to decide when to save → no issues.

- You write code to serialize your state and parse the saved file.
- You save the strings.
- What could possibly go wrong?

How to save the state of your application?

90's - It was quite easy. User pressed the save button. No shared state, single threaded, no need to decide when to save → no issues.

- You write code to serialize your state and parse the saved file.
- You save the strings.
- What could possibly go wrong?
- Some care still needed. E.g. copy the file, rewrite it and then delete the copy.
- The user should not turn the computer until this process finishes.

We need consistency and durability.

Data exchange formats [1]

BTW, how to serialize data?

- Markup languages: XML, JSON, YAML,
- Portable, extensible solution (unless you really need to send a lot of data).

Persistence

Today, we often want to save changes as they appear.

- When to save changes?
- We do not want to save a state where we removed money from one account and did not add money to the other

We need: atomicity.

Persistence

In addition: we work on more things concurrently.

- We could try to manage discrepancies in data, but we would prefer not do this as this might get very complex.
- Ideally each concurrent tasks should look like it is the only task being done right now.

We need: isolation.

ACID [2]

- Atomicity - Guarantees that each transaction is treated as a single "unit", which either succeeds completely, or fails completely.
- Consistency -A transaction can only bring the database from one valid state to another, maintaining database invariants.
- Isolation - Concurrent execution of transactions leaves the database in the same state that would have been obtained if the transactions were executed sequentially.
- Durability - once a transaction has been committed, it will remain committed.

These properties of database transactions should hold even in the event of crashes power failures, etc.

ACID Database

- There is a lot of complexity associated with persistence (and related, now almost unavoidable, concurrency)
- Strong ACID makes reasoning about what is going on much easier.
- A lot of complexity is transferred to the database implementation.
- The guarantees come at a performance cost (the cost heavily depends on what you do).
  - This performance hit can be especially troublesome for distributed databases - any cooperation between nodes requires a network call.
- The guarantees are way stronger than the requirements for most business rules.

< ACID

How to relax atomicity and consistency:

- Atomicity provided only within a smaller scope (aggregate, one table).
- You cannot create / delete elements together with other changes.

How to deal with it:

- You can simulate transactions in a similar way how you split transaction when you wait for I/O.
- In many use cases you can accept a reasonable risk of creating inconsistent state and then try to resolve it.

< ACID

Isolation:

- Lost update - two transactions concurrently update the same data, one of them is ignored due to a race condition.
- Dirty reads - we read uncommitted data.
- Non-repeatable reads - we read the same element twice with different results (because some other transaction finished).
- Phantom reads - new rows are added or removed by another transaction to the records being read.

Isolation levels:

- Serializable
- Repeatable reads
- Read committed
- Read uncommitted

< ACID

Durability example, MongoDB - write concerns (there are also read concerns):

- Unacknowledged - Write operations that use this write concern will return as soon as the message is written to the socket.
- Acknowledged - Write operations that use this write concern will wait for acknowledgment from the primary server before returning.
- Journaled - Write operation waits for the server to group commit to the journal file on disk.
- Replica Acknowledged (two / majority) - Waits for at least two / majority of servers for the write operation.

< ACID

Durability:

- If there is something super important, your write concern is Replica Acknowledged majority and your read concern requires you to check majority of replicas.
- Doing stuff super save → big response time, long locks (and thus small throughput).
- Note that to guarantee that something is written into majority of replicas requires significant networking (maybe in seconds).
- If the change requires a lock with larger scope, nobody else can modify that scope.
- The system may only manage several such writes each minute.

ACID vs BASE [4]

- Basically Available: basic reading and writing operations are available as much as possible (using all nodes of a database cluster), but without any kind of consistency guarantees (the write may not persist after conflicts are reconciled, the read may not get the latest write).
- Soft state: without consistency guarantees, after some amount of time, we only have some probability of knowing the state, since it may not yet have converged.
- Eventually consistent: If the system is functioning and we wait long enough after any given set of inputs, we will eventually be able to know what the state of the database is, and so any further reads will be consistent with our expectations.

There are various reason to get into an inconsistent state i.e.:

- concurrent writes (e.g. in distinct nodes)
- preferring availability and act despite some node is unavailable.

ACID vs BASE [4]

To converge to eventually consistent state, we must

- exchanging versions or updates of data between servers.
- choosing an appropriate final state when inconsistency is detected (reconciliation):

Reconciliation options include:

- last writer wins,
- user specified conflict handler,
- first writer wins.

Causally consistency

Soft state may be inconvenient - if you make an action and the result disappears for a while (the write has not spread to the replica you read from), it is very confusing for the user. One can deal with this problem either on client side or on server side:

- Causally consistent sessions:
  - Read Your Writes
  - Monotonic reads
  - Monotonic writes
  - Writes follow reads
- The application can e.g. cache its writes so if it requires to read previously written value it uses its cached value instead.

Base examples

Conflicting price of a product:

- What if we read an old value (new value yet unavailable in our replica): No problem, this implies a very recent price change. From the business point of view we can consider the order to be done few seconds earlier. The difference is mostly irrelevant.
- Inconsistent values due to concurrent update: Last writer wins seems mostly fine.
- Inconsistent values due node unavailability: Last writer wins seems fine.

Base examples

Content of the cart:

- Custom handler: union.

What is the amount on my account?

- We need to implement locking. We reconciliate locking issues very defensively.

CAP theorem

CAP theorem [6] - we cannot achieve all consistency, availability, and partition tolerance (however they are defined) in asynchronous network model.

Besides this, the name CAP theorem is also used to describe the following simple observation.

- Assume that some node is unavailable (e.g because there is no link to that node), we get a request.
  - We can postpone processing the request until the node becomes available again (no partition tolerance).
  - We can respond that we cannot handle the request (no availability).
  - We try to handle the request, but we cannot guarantee that we remain consistent with the unavailable node (no consistency)

CAP theorem

Example: two nodes storing one number with one invariant: the number must be the same in both nodes.

- If a request arrives to change the number but due to broken link, we cannot communicate with other node, if we want to respond (partition tolerance) and preserve the invariant (consistency), we have to refuse the change (no availability).

For practical purposes, it is about the balance of availability vs consistency.

- **Where the balance is is not a technical decision but a business decision.**

Availability vs consistency

The business often does not require 100% consistency.
Overbooking example:

- You want to reserve a room, but the server in charge of handling the reservation is not available.
- Can I process the request if 10 minutes ago 10 rooms were available (an we rarely get more than one request per day).
    - If we respond, that we cannot process requests currently, customer may find other accommodation .
    - It is important for the business to be available, even at the cost of consistency.
    - We should probably proceed even if only 1 room is available, in case of overbooking there are various ways to handle it.
    - In some businesses overbooking is standard, expecting cancellations (demonstrates what a non-issue consistency here is).

As you see often availability >> consistency.

Availability vs consistency

Often availability $>>$ consistency. This applies even if other nodes are available.

- Nobody likes working with slow apps.
- Instead of asking the server about the number of available rooms, we can decide, it is save to assume that the room is available and skip the network call.
    - Especially if we have a very recent replica read.

Sometimes latency $>>$ consistency.

Distributed databases

There are two very distinct ways to distribute a database (those approaches can be combined):

- Replication - storing separate copies at two or more nodes
  - Single leader - one server receives writes.
  - Multi leader, Leaderless
- Fragmentation - we divide into smaller parts and then store them on separate nodes
  - Horizontal fragmentation - e.g. split the rows of the table
  - Vertical fragmentation - e.g. split the columns of the table (primary key in both tables)

Single leader replication is the most common solution as writes are usually much less frequent than reads.

Databases: Underlaying data structure

Databases can be based on various data structures, i.e.:

- Key-value pairs: stores data as values accessed by keys.
- Documents: stores data as documents accessed by keys.
  - Values in key-value pairs are opaque, documents are transparent (thus we can index fields within document).
- Wide column store: stores data in rows in tables, but columns are not prescribed.
  - May be interpreted as 2-dimensional key-value store.
- Relational tables: stores data as rows in relational tables.
- Graphs: stores data as labeled vertices and edges of a (directed) graph.

Databases: Underlaying data structure

Relational:

- Well-known
- Standardized and widely used query language - SQL.
- R&D $>$ 40 years.
- Offer great deal of flexibility.

It is a good default choice. We compare other database types with relational databases.

Key-value / document / wide column

In all cases you put related data into something: value / document / row - we call it an aggregate. Thus these choices are surprisingly similar in the end.

- It is easy to read and modify the aggregate.
  - Relational database may need to do several joins to obtain the data (but still doable thanks to relational database flexibility).
  - This includes acquiring the necessary locks.
- It is inefficient to query something that spans many aggregates.
  - This may include acquiring the necessary locks, a lot of locks.
- Aggregates define boundaries that help to distribute the data more efficiently.

Key-value / document / wide column

Example: Order contains several lines that contain item id, name, price and quantity. We can store it as a document or as several rows in a relational table (or tables). First, we want to get the order.

- It is easy to read the order.
  - In relational database we may need to read several rows from several tables.

Now we want to get how many items with id `id` we sold yesterday:

- We have to read all aggregates from yesterday. That may be a lot of data.
  - In relational database we still have to select some rows and put them together.

Access is excellent if it is within aggregate, but may become prohibitively hard if it involves many aggregates. Relational databases work reasonably well in both cases.

Graph databases

We show what are graph databases good for by example. We want to select an advertisement for a product of similar type as user looked at in a shop recently.

- We might have relational tables like this: `Visits(User, Page)`, `PageCategory(Page, Type)`, `PageContains(Page, Product)`, `ProductIs(Product, Category)`, `SimilarCategories(Product, Product, Similarity)`, `AdvertisementContains(Advertisement, Product)`
- `SELECT ...` not easy to write, even harder to execute, potentially huge locks necessary.
- A simple graph traversal problem, quite easy to write in appropriate query language, not that hard to execute.
- This is, indeed, one of the typical use-cases of graph databases.

To put things together

Database type	Document	Relational	Graph
data stored as	document	row	vertex, edge
fragmentation	→ more fragmented data →		
queries	→ more flexibility →		
ACID	→ more need for ACID transactions →		

But

- If you can afford to use ACID transactions, you probably want to use them.
- Relational databases are typically the safe choice.

Resources I

- Wikipedia - ACID
- Wikipedia - Isolation
- Wikipedia - Eventual consistency
- Video: M. Fowler: Introduction to NoSQL

References I

- Wikipedia - Data exchange formats
- Wikipedia - ACID
- Wikipedia - Isolation
- Wikipedia - Eventual consistency
- Understanding Durability & Write Safety in MongoDB
- S. Gilbert, N. Lynch: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services