ACID vs. BASE

Ondra Chaloupka / ochaloup@redhat.com

Why this presentation?

Distributed systems changed the way how we process data and way how we can think about transactions. This could give you some summary about the topic.

What you will get in this 30+ minutes presentation?

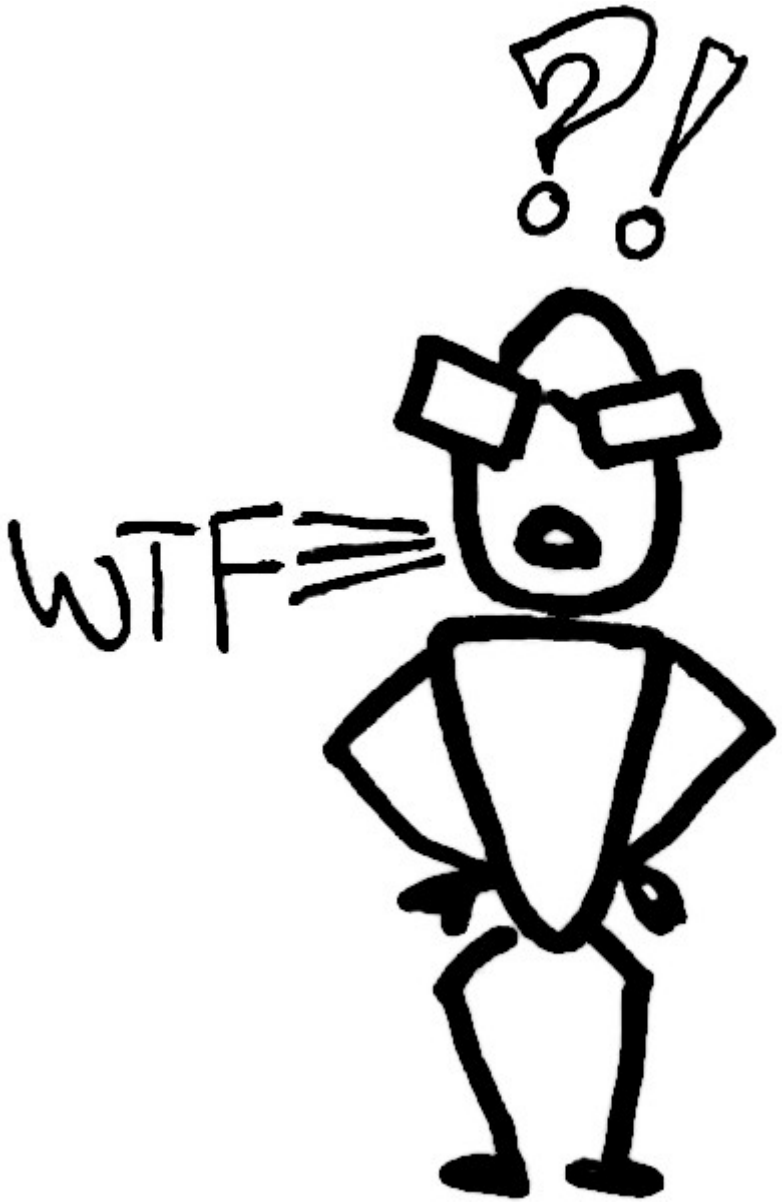
NOTE

Get to know nowadays transaction buzzwords and get some knowledge what you can use when need to search the Internet :)

Disclaimer: the presentation could contain some simplifications, for more details, please, check the topic on your own

There are quotes coming from different places on the Internet in the presentation. All links are mentioned.

Why transactions?



NOTE

- Transactions providing programming model which ease tasks of data manipulation.
- Transactions provides guarantees which you can build upon (anybody would be happy having assurance that a data change doesn't break data integrity)

ACID

- **A** for atomicity
- **C** for consistency
- **I** for isolation
- **D** for durability

Transactions are usually connected with acronym ACID which defines set of properties of a (database) transaction.

Atomicity

"all or nothing", all operations in a transaction succeed or every operation is rolled back

Consistency

on the completion of a transaction, the database is structurally sound that covers e.g. preserve foreign keys, uniqueness defined by schema etc.

Isolation

transactions do not contend with one another. Contentious access to data is moderated by the database so that transactions appear to run sequentially.

Durability

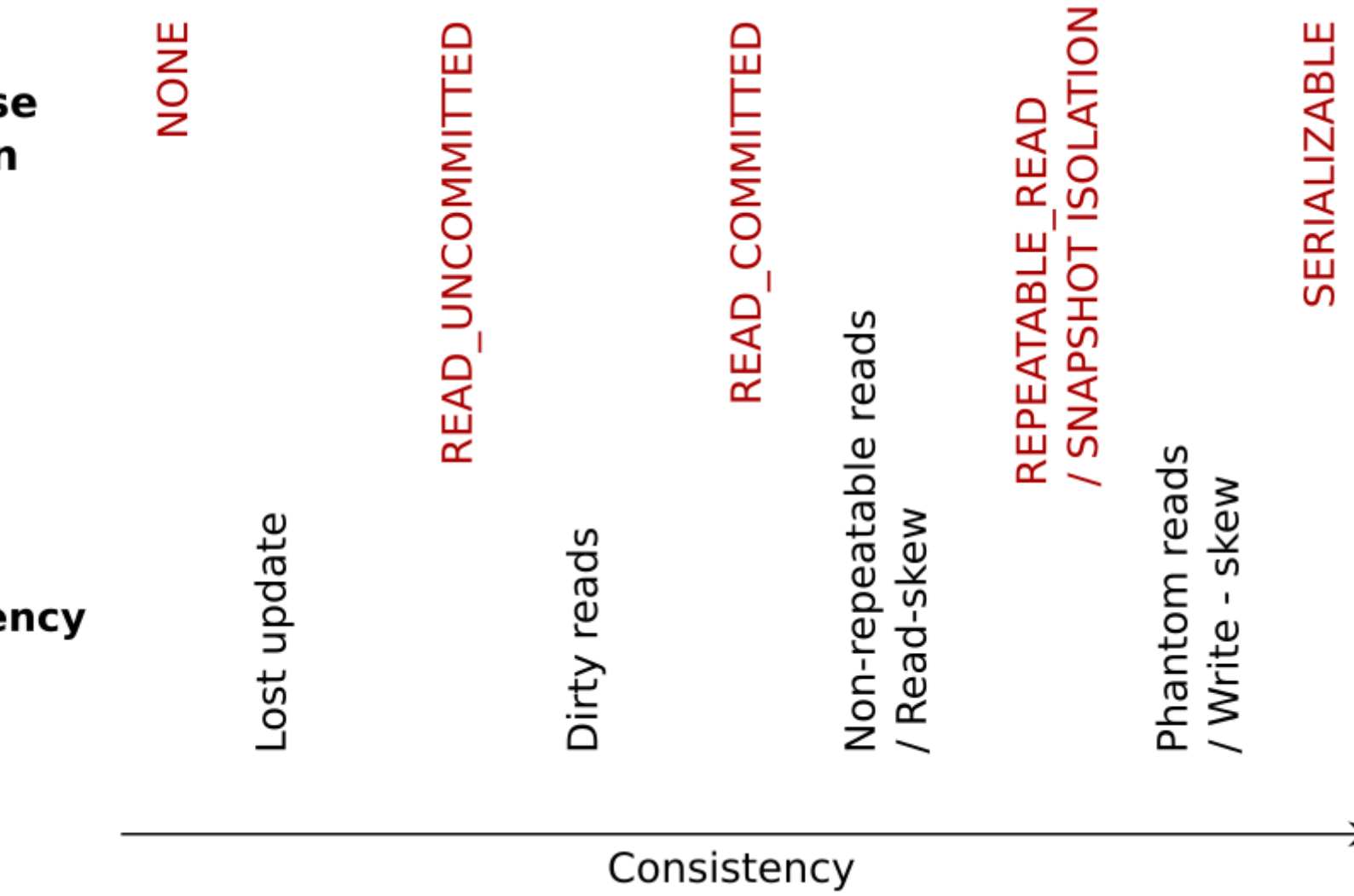
the results of applying a transaction are permanent, even in the presence of failures

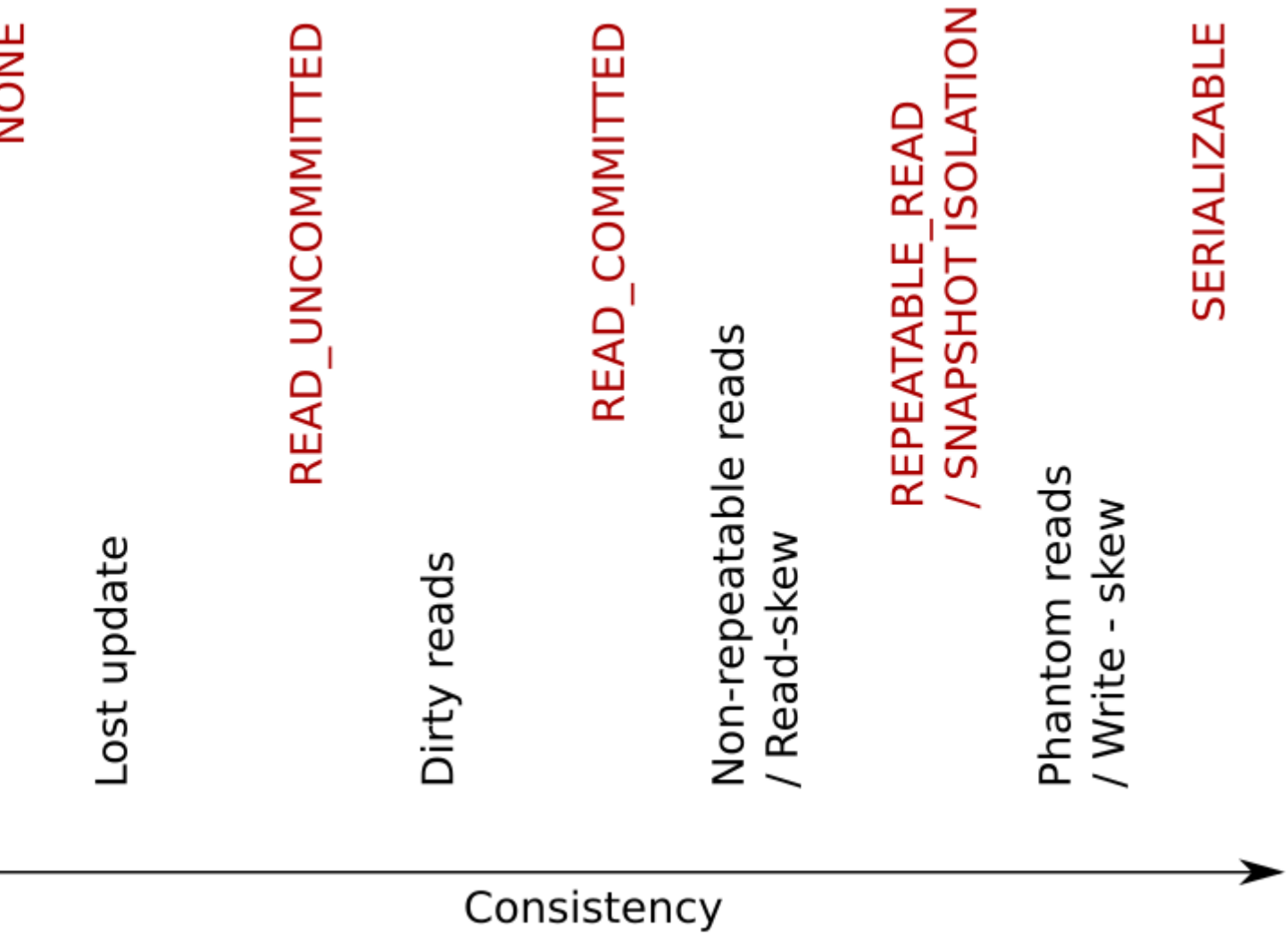
The ACID acronym came from research of `IBM System R` from year 1975. Acronym itself mixes different standpoints about a system (more mnemonic than precise, Brewer 2012) Following description came from Martin Kleppmann's presentation linked below.

NOTE

- **Durability** means the time when disk does `fsync` (leave some deeper technical details about disk writes aside). For transaction it means that data are written to some log and when system crashes it will be available when started again (we can re-read the log and restore data).
- **Atomicity** defines possibility to abort transaction and changes done by transaction in the system will be reverted to state before the transaction starts. Martin talks that it should be nicer to says `abortability`.
  Atomicity is about handling failures when does depend when they come from (system crash, network failures, some constraint was broken…).
  Atomicity is **not** about concurrency. Rather *Isolation* is about concurrency - meaning parallel transaction works on the same piece of data.
- **Consistency** is to having the system in consistent state (moving from one to other one). *Consistency* is a point of view of an application in fact.
  Consistency means fulfilling invariants which could be defined in DB model but they could be outside of it too.
- **Isolation** is about concurrency. In perspective of this presentation the *Isolation* property is the most interesting. When talking about *ACID* isolation it's meant *Serializability* to be obtained.
- https://en.wikipedia.org/wiki/ACID
- https://martin.kleppmann.com/2015/11/04/transactions-at-code-mesh.html

Isolation


NONE
READ_UNCOMMITTED
READ_COMMITTED
REPEATABLE_READ
/ SNAPSHOT ISOLATION
SERIALIZABLE
Lost update
Dirty reads
Non-repeatable reads
/ Read-skew
Phantom reads
/ Write - skew
Consistency



Isolation talks about behavior when running two (multiple) concurrent transaction. We know classic isolation levels defined by ANSI/ISO SQL standard. Those are based on notion of locking (lock-based concurrency control). I'm mentioning locks in the list belows but it's just naive way of thinking about locking. Real DBs has much more sophisticated ways to lock thinks (e.g. check M$ articles about locks)

None

- locks: no
- struggles *dirty writes (P0)*
- you can read corrupted data that as other transaction hasn't ended with an operation yet
- means that your not committed work could be rewritten by other "transaction" meanwhile.
- here is interesting point about *lost update (P4)* phenomena
  you can read at some places that the minimum transaction isolation level to avoid it is *read uncommitted*. Which is partly true. The issue (as I understand it) is that the term *lost update* is not precisely specified. For *read uncommitted* solves this phenomena it has to mean that other transaction can interleave writing a data which can lead to corruption of data (no lock acquired, two transaction writing to the same place at the same time)

- but *lost update (P4)* is specified rather differently (more broader), fist or second
- data that has been updated by one transaction is overwritten by another transaction, before the first transaction is either committed or rolled back
- one transaction reads data into its local memory, and then another transaction changes this data and commits its change. After this, the first transaction updates the same data based on what it read into memory before another transaction was executed. In this case, the update performed by the another transaction can be considered a lost update.
- the later trouble connects to contention errors too (deadlocks) and it's a thing which Hibernate (ORM) tries to solve (entities are loaded to cache), as solution you can then using optimistic locking (version added to data and exception thrown when concurrent change happens) or pessimistic (`select ⋯ for update` is used which causes acquiring exclusive write lock on accessed rows)

Read uncommitted

- struggles *dirty reads (P1)* phenomena
- locks: acquires write lock only for the operation and releases immediately
- means you can read uncommitted changes of a different transaction. But you can't write to data which was changed by some concurrent transaction.

Read committed

- struggles *(fuzzy) non-repeatable reads (P2)* phenomena
- locks: acquires write lock to the end of transaction, read lock is released immediately after select ends
- means that other transaction reading the same data several times can see different values during it's executions. Transaction one starts and reads X being 1. Meanwhile transaction two stats, writes X to be 2 and commits. Transaction one reads X again and it can see it being 2.

Repeatable reads

- struggles *phantom reads (P3)* phenomena
- locks: acquires write and read lock for records working with till the end of transaction
- means that reading a set of data over a table could change during life time. Transaction one starts and selects `SELECT * from MYTABLE`, Transaction two starts, insert a row to the table `MYTABLE` and ends. Transaction one selects `MYTABLE` again and it can see different number of results now.

Serializable

- 'complete' isolation of transaction
- locks read, write and range lock when select is used are acquired till the end of transaction

Now we can see there is not only those isolation levels but the diagram shows **Snapshot isolation (MVCC)**. That's a different approach of solving the concurrency control issue. We do not use locks but a snapshot of data is taken at time when transaction starts. Transaction then works with the data which was available at point of transaction started. Still for being sure to not get issues on writing data we need to acquires write locks (but we don't need read locks). Acquiring locks is a pesimistic way of solving the issue, the optimistic one is to record writing and in case of conflicting abort the transaction and retry it again. In systems with no much contentions it works fine. See more about **Serializable Snaphost Isolation** e.g. at PostgreSQL web.

Let's shortly mention other two more concurrency control issues which connect to snapshot isolation. The schema on this slide is not exact as lock based repeatable read avoids *non-repeatable reads* and *read skew* (which could be taken as special case of *non-repeatable reads*) but it avoids *write skew* too which is not avoided by *snapshot isolation* (when not talking about *SSI*).

Read Skew *(A5A)*

- not avoided when using *read committed* isolation level, avoided when *snapshot isolation* is used and when *repeatable read* is used
- variation on *non-repeatable reads*. When transaction 1 selects record a, transaction 2 sneaks and updates the record 1 and record 2, transaction 1 resumes and selects record 2. Transaction 1 does not see the picture of the world as it was when it starts (when it read record 1).

Write Skew *(A5B)*

- avoided when using *repeatable read* isolation level (based on locks), not avoided when *snapshot isolation* is used. Occurs when transaction are not *serializable*.
- it defines situation of some constraint being put at application level. For example that at least one counter in a table has to be non-zero (Martin makes example in his presentation on at least one physician has be in attendance and all decide that decline his attendance in the same time). There is important here that two transactions operates on different records thus it does not influence each other directly on write.
  When back to counter we have this DB table:

ID	Counter
1	1
2	1

Now transactions runs concurrently

Transaction 1	Transaction 2
BEGIN txn 1	BEGIN txn 2

Transaction 1	Transaction 2
SELECT * FROM table	SELECT * FROM table
check there if there is some other record with counter >= 1	check there if there is some other record with counter >= 1
set counter of **ID 1** to **0**	set counter of **ID 2** to **0**
COMMIT	COMMIT

At the end both counters (counter of ID = 1 and ID = 2) are zero which is violation of constraint application has.

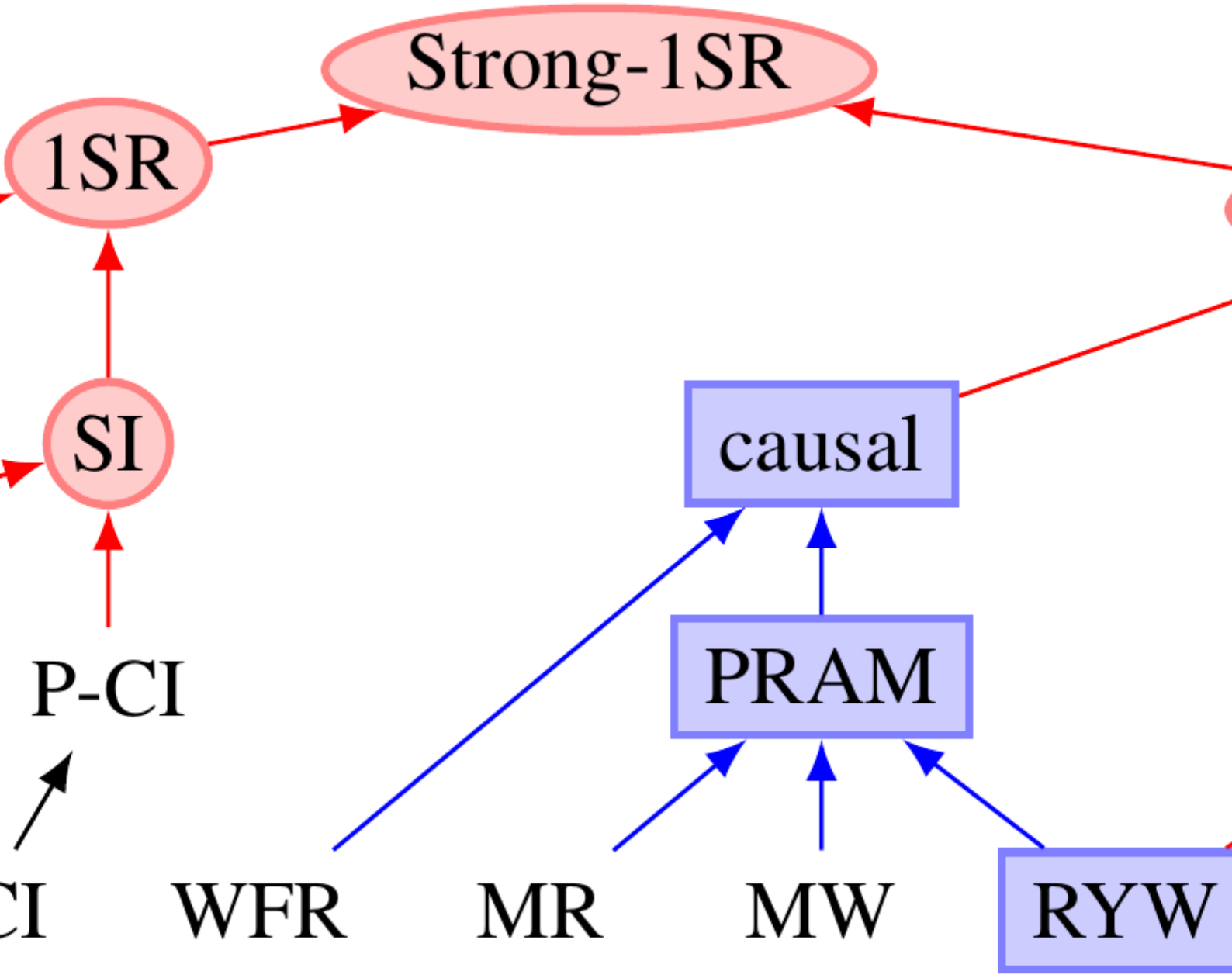
- https://martin.kleppmann.com/2014/11/25/hermitage-testing-the-i-in-acid.html
- http://blog.triona.de/development/database/acid-and-isolation-level-overview.html
- http://ithare.com/databases-101-acid-mvcc-vs-locks-transaction-isolation-levels-and-concurrency
- https://technet.microsoft.com/en-us/library/jj856598
- http://technet.microsoft.com/en-us/library/cc546518.aspx
- https://www.simple-talk.com/sql/t-sql-programming/developing-modifications-that-survive-concurrency
- https://wiki.postgresql.org/wiki/SSI
- https://vladmihalcea.com/2015/10/20/a-beginners-guide-to-read-and-write-skew-phenomena

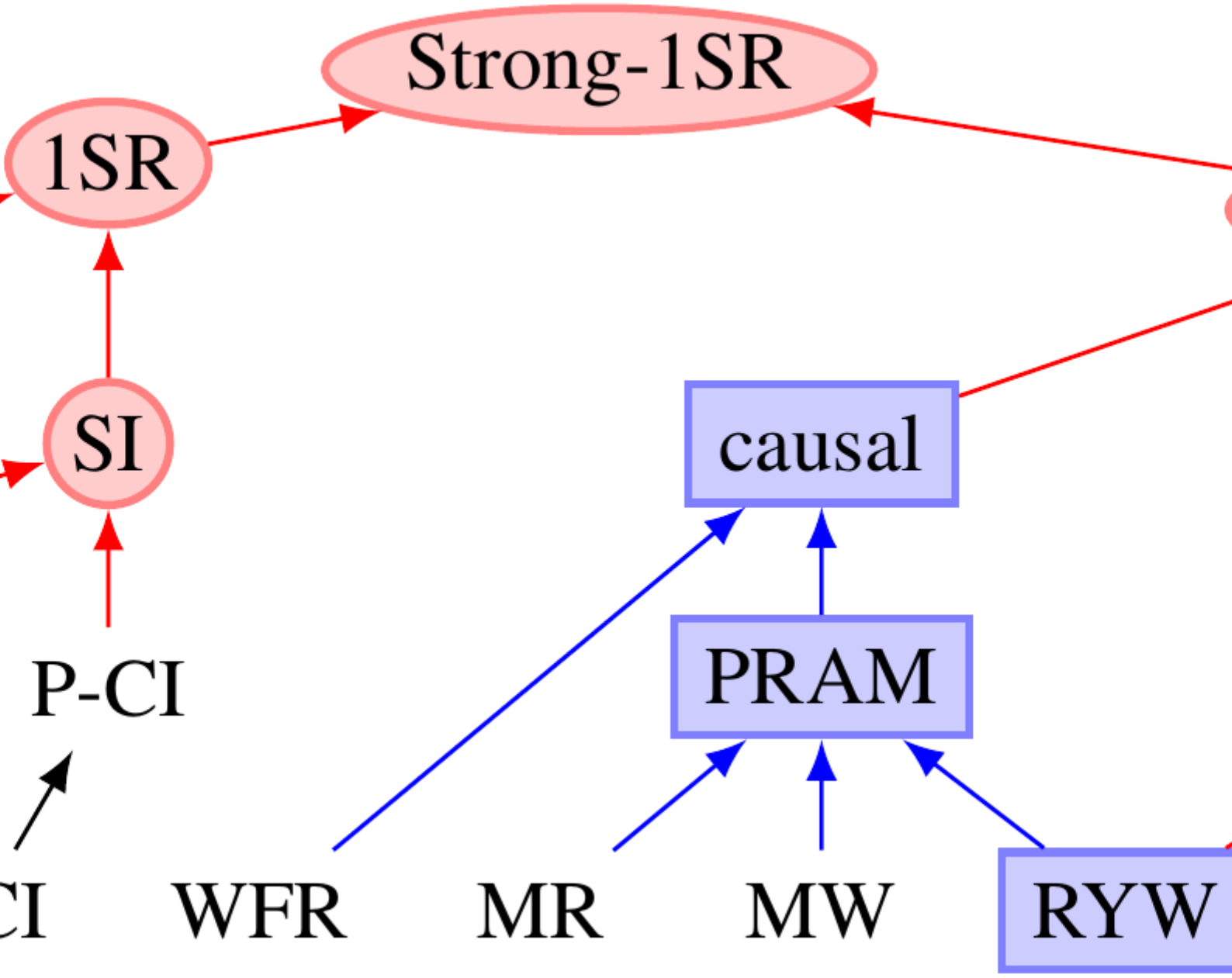
Isolation in real world

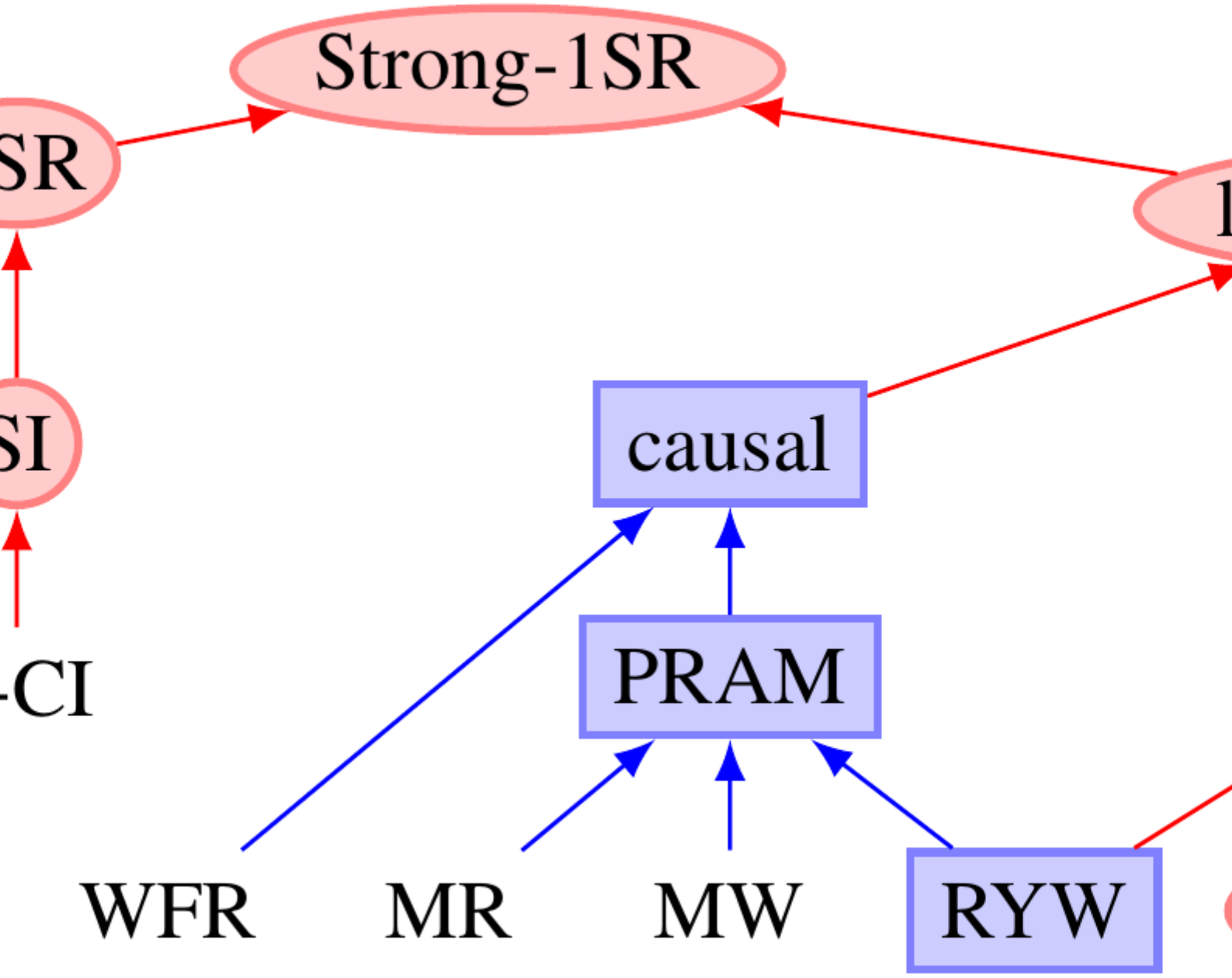
- https://github.com/ept/hermitage
- http://www.bailis.org/blog/when-is-acid-acid-rarely

NOTE Links contains tables where are depicted default and maximum isolation levels for particular databases.

Isolation for distributed world


Strong-1SR
1SR
SI
causal
P-CI
PRAM
CI
WFR
MR
MW
RYW


Strong-1SR
1SR
SI
causal
P-CI
PRAM
CI
WFR
MR
MW
RYW



Some more info about distributed DBs and HA in a while but here we can see a picture of Peter Bailis paper *HAT, not CAP: Introducing Highly Available Transactions*.

- It presents what are isolation levels available for HA (AP from CAP) systems.
- Those marked as red are not available for CA.
- Those marked in blue are available for sticky availability.
- Those which are on the left side of the picture (RU, RC, RR, SI, 1SR) are those discussed in previous slide. The part on the right side belongs to the distributed world that we talk about in a while.

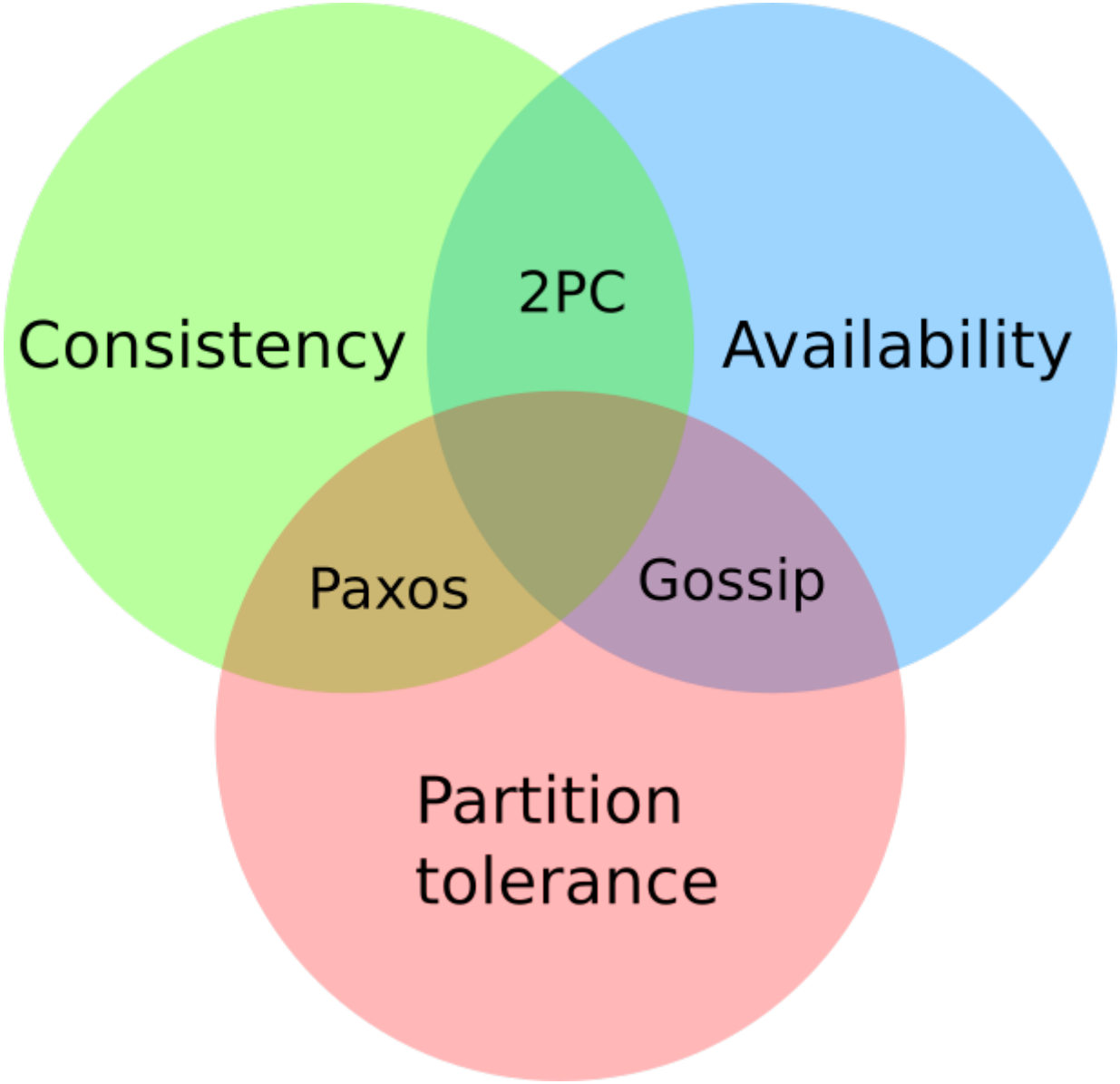
HA	Read Uncommitted (RU), Read Committed (RC), Monotonic Atomic View (MAV), Item Cut Isolation (I-CI), Predicate Cut Isolation (P-CI), Writes Follow Reads (WFR), Monotonic Reads (MR), Monotonic Writes (MW)
Sticky	Read Your Writes (RYW), PRAM, Causal
Unavailable	Cursor Stability (CS)† , Snapshot Isolation (SI)† , Repeatable Read (RR)†‡ , One-Copy Serializability (1SR)†‡, Recency , Safe , Regular , ⊕Linearizability , Strong 1SR†‡

I would like quickly touch one topic here which is Serializability vs. Linearizability. It's interesting from point that even serializable isolation level can't be taken as the level of the most guarantees. Still there is a *Strict Serializability*.

To quote Peter Bailis here

- **Linearizability** is a guarantee about single operations on single objects. It provides a real-time (i.e., wall-clock) guarantee on the behavior of a set of single operations (often reads and writes) on a single object (e.g., distributed register or data item).
  - *Linearizability* for read and write operations is synonymous with the term `"atomic consistency"` and is the `"C"`, or `consistency`, in Gilbert and Lynch's proof of the *CAP Theorem*.
- **Serializability** is a guarantee about transactions, or groups of one or more operations over one or more objects. It guarantees that the execution of a set of transactions (usually containing read and write operations) over multiple items is equivalent to some serial execution (total ordering) of the transactions.
  - *Serializability* is the traditional `"I"`, or `isolation`, in *ACID*.
- http://www.bailis.org/blog/hat-not-cap-introducing-highly-available-transactions
- http://www.bailis.org/blog/linearizability-versus-serializability
- https://blog.acolyer.org/2016/02/26/distributed-consistency-and-session-anomalies
- https://www.youtube.com/watch?v=Ih0Efbx0cE8 : Adrian Colyer - Out of the Fire Swamp

CAP



(*Coined by* `Dr. Eric Brewer` *by talk* `Towards Robust Distributed Systems` *in 2000. Seth Gilbert and Professor Nancy Lynch formalized in 2002.*)

The CAP Theorem (henceforth 'CAP') says that it is impossible to build an implementation of read-write storage in an asynchronous network that satisfies all of the three properties. We are constrained only to two of them.

Availability

will a request made to the data store always eventually complete

Consistency

will all executions of reads and writes seen by all nodes be atomic or linearizably consistent

Partition tolerance

the network is allowed to drop any messages.

It's a popular and fairly useful way to think about tradeoffs in the guarantees that a system design makes.

In *normal* distributed system we can't take off **P** - we are limited for **CP** or **AP**.

The diagram shows **CA** as **2PC**. It's possible in way that we avoid partition to happens by not using distributed execution, by running on single node. Then we got to well known XA distributed transactions 2PC aka. ACID.

And hey, wait a minute, this talk about `read-write storage` and not any transaction ;)

Let's revise the **CAP** acronym once again

Partition-Tolerant environment

ability of the whole system to continue to work (accept requests and process them reliably) even when any number of messages fail in communications.

- the basic prove of **CAP theorem** is based on having two nodes split (brain split), each of them starts to process different requests but they can't communicate with each other. The system as whole is then inconsistent (consistency in CAP way of thinking). Each node responses different value on reading.
- Consistency has multiple forms - CAP talks about linearizability (strict consistency)

Consistency

defines a consistency model that system work with. the characteristic is whether every read would return the latest written information (or error), it's a degree of how soon the changes done by your transaction is visible to other transactions

- Very often people attempting to introduce eventual consistency into a system run into problems from the business side. Business users hear "consistency" and they tend to think it means that the data will be wrong. That the data will be incoherent and contradictory.)

Availability

excects that every request receives a non-error response

- Cloud providers have broadened the interpretation of the CAP theorem in the sense that they consider a system to be unavailable if the response time exceeds the latency limit.
- CAP talks about total availability
  - https://henryr.github.io/cap-faq
  - http://book.mixu.net/distsys/single-page.html
  - https://medium.com/@cinish/database-acid-cap-isolation-levels-371b7e06a112
  - https://msdn.microsoft.com/en-us/library/jj591577.aspx
  - TODO review: http://www.cs.utexas.edu/~dsb/cs386d/Projects14/CAPConsistency.pdf

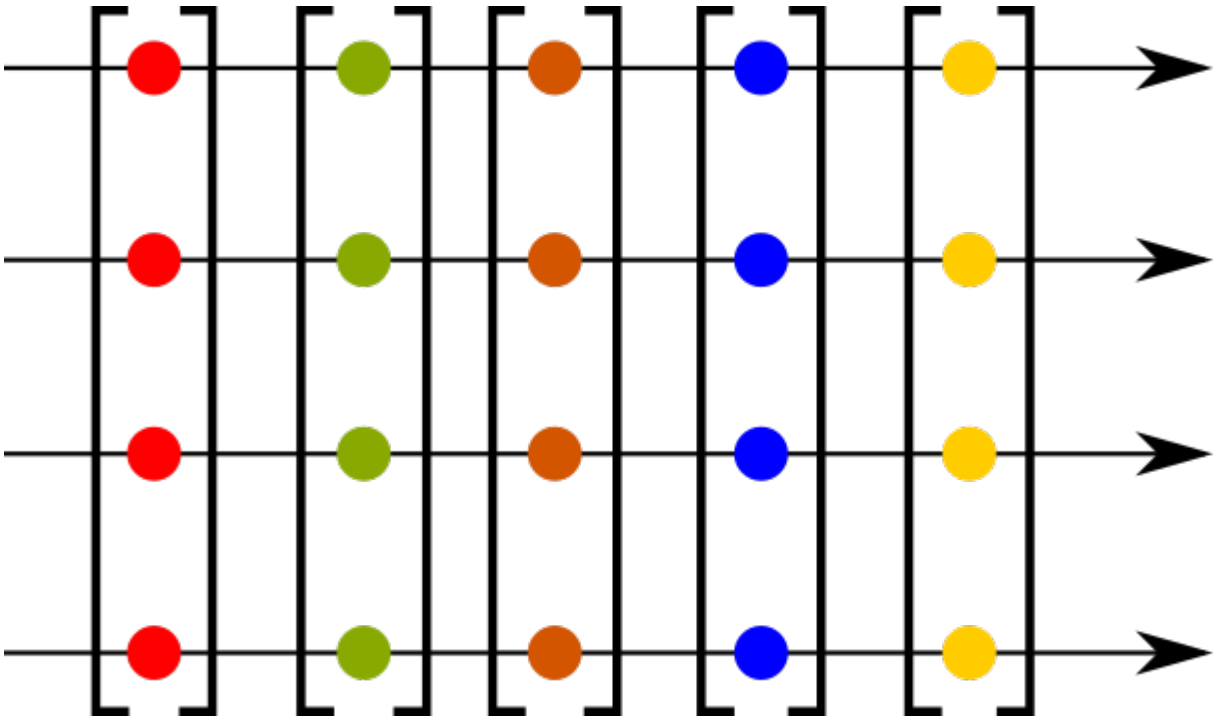
Serializability

NOTE

Serialization talks about multiple items groups under one transaction. Those transaction can be put to arbitrary order but they are executed in serial.

Easy to imagine is to run on one node in one thread transaction by transaction.

Linearizability

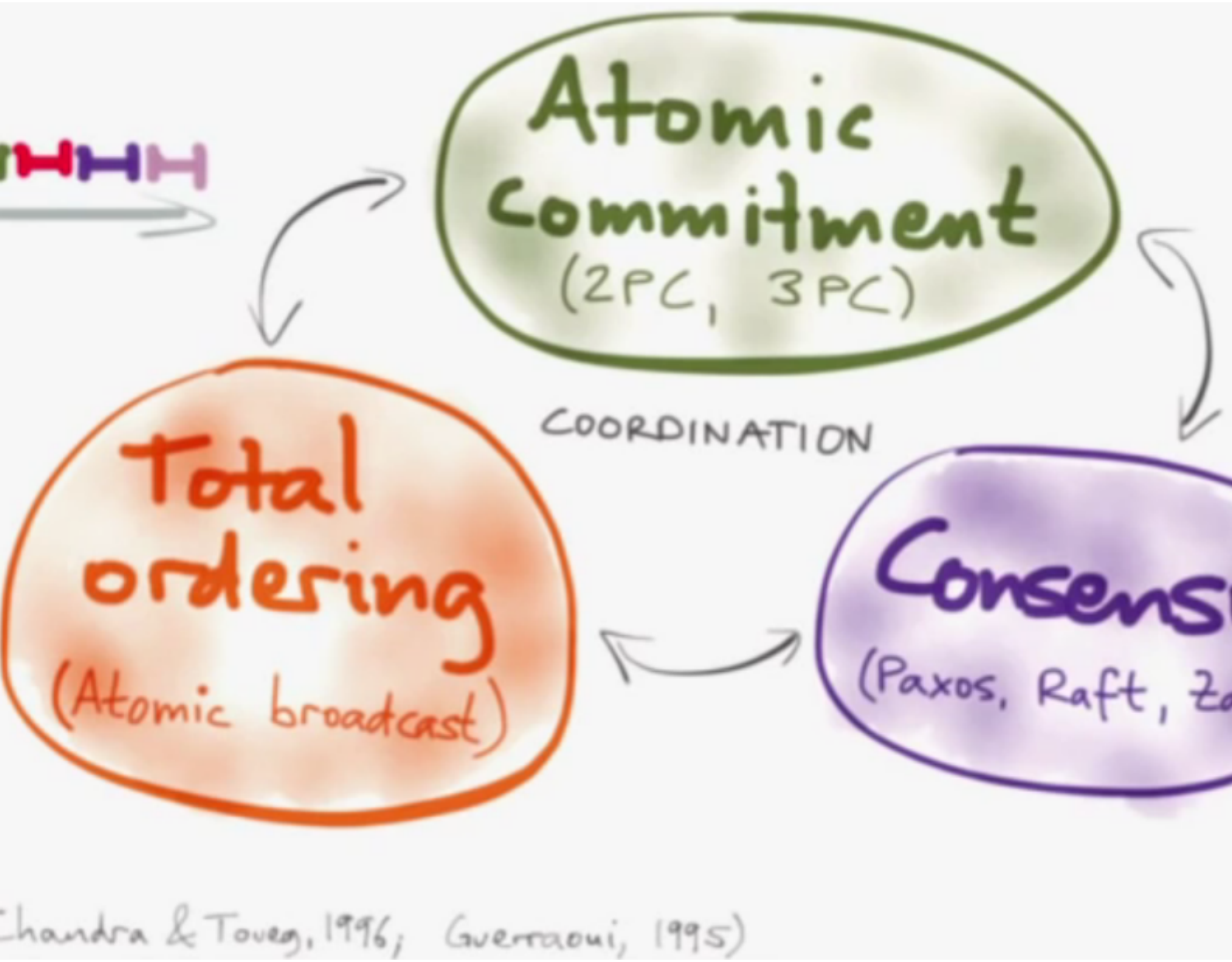


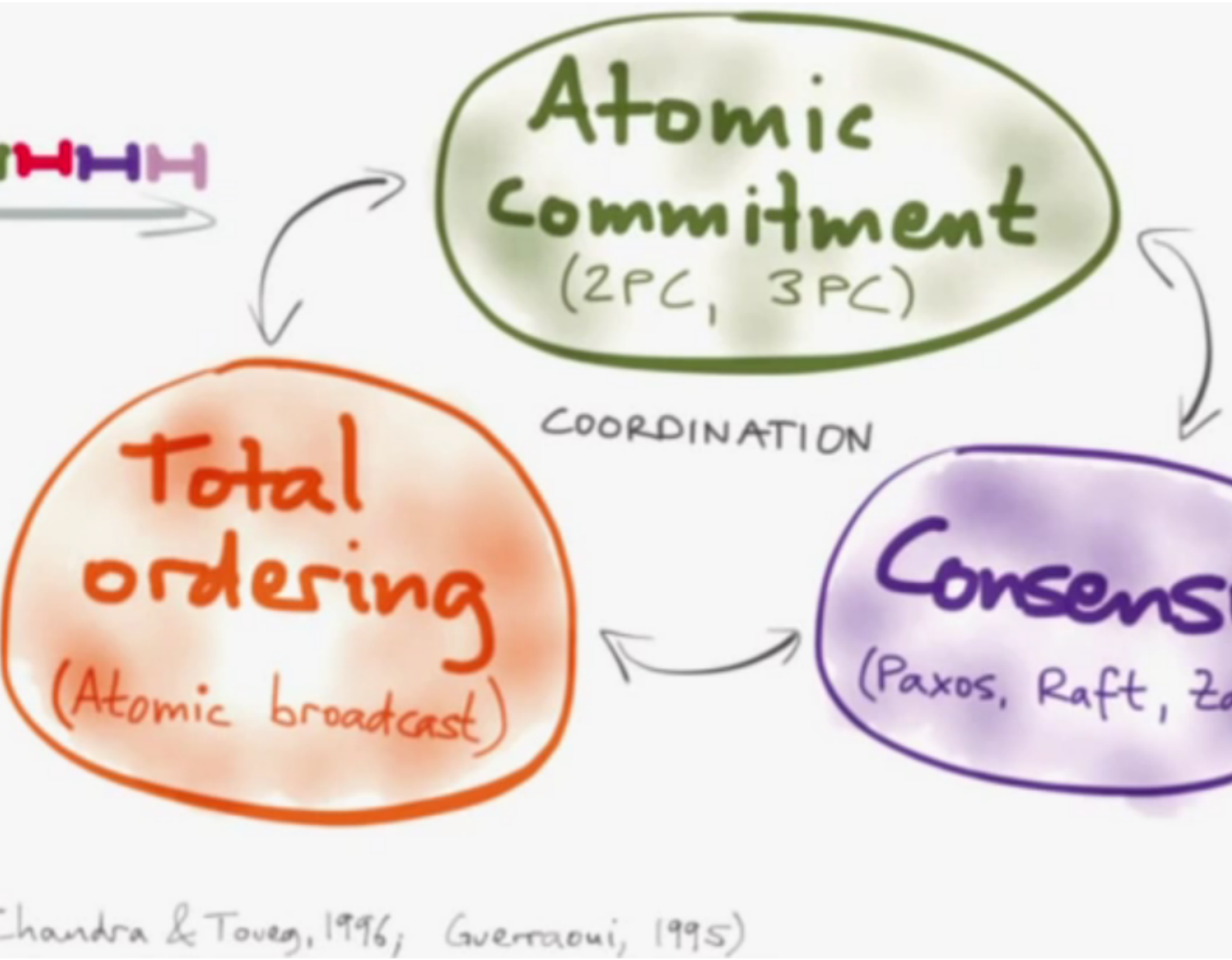
NOTE

Linearizability talks operation on single object to be done in the same order on multiple nodes.

In other words it ensures that all operations on single object is executed in the same order on all nodes.

Distributed serializability





source: Martin Kleppmann

> Here I would like check with you what are things to achieve **serializable transactions** in distributed environment. For that we need an **atomic commitment protocol** (all of them protocol or nono of them commit) and for that work in distributed environment we are in fight with **ordering**. And total ordering is in fact a problem of **consensus**.
>
> Consensus requires quite a lot of coordination.
>
> Possible uses of consensus are:
>
> - deciding whether or not to commit a transaction to a database
> - synchronising clocks by agreeing on the current time
> - agreeing to move to the next stage of a distributed algorithm (this is the famous replicated state machine approach)
> - electing a leader node to coordinate some higher-level protocol

Several computers (or nodes) achieve consensus if they all agree on some value. More formally:

1. Agreement: Every correct process must agree on the same value.
2. Integrity: Every correct process decides at most one value, and if it decides some value, then it must have been proposed by some process.
3. Termination: All processes eventually reach a decision.
4. Validity: If all correct processes propose the same value V, then all correct processes decide V.

A step aside: FLP

FLP talks on problem of consensus

```
Having all nodes agree on a common value - is unsolvable
in general in asynchronous networks where one node might fail.
```

This is only a side node to the fact that scientists considers different models to describe distributed systems and the '*CAP issue*'.

- FLP permits the possibility of one 'failed' node which is totally partitioned from the network and does not have to respond to requests.
- Otherwise, FLP does not allow message loss; the network is only asynchronous but not lossy.
- FLP deals with consensus, which is a similar but different problem to atomic storage.
- https://blog.acolyer.org/2015/09/02/the-potential-dangers-of-causal-consistency-and-an-explicit-solution
- http://book.mixu.net/distsys/single-page.html
- https://aphyr.com/posts/322-call-me-maybe-mongodb-stale-reads

Consistency in baseball game

	1	2	3	4	5	6	7	8	9	RUNS
Visitors	0	0	1	0	1	0	0			2
Home	1	0	1	1	0	2				5

ng Consistency	2-5
ntual Consistency	0-0, 0-1, 0-2, 0-3, 0-4, 0-5, 1-0, 1-1, 1-2, 1-3, 1-4, 1-5, 2-0, 2-1, 2-2, 2-3, 2-4, 2-5
sistent Prefix	0-0, 0-1, 1-1, 1-2, 1-3, 2-3, 2-4, 2-5
nded Staleness	scores that are at most one inning out-of-date: 2-3, 2-4, 2-5
notonic Reads	after reading 1-3: 1-3, 1-4, 1-5, 2-3, 2-4, 2-5
d My Writes	for the writer: 2-5 for anyone other than the writer: 0-0, 0-1, 0-2, 0-3, 0-4, 0-5, 1-0, 1-1, 1-2, 1-3, 1-4, 1-5, 2-0, 2-1, 2-2, 2-3, 2-4, 2-5

	1	2	3	4	5	6	7	8	9	RUNS
	0	0	1	0	1	0	0			2
	1	0	1	1	0	2				5

2-5
0-0, 0-1, 0-2, 0-3, 0-4, 0-5, 1-0, 1-1, 1-2, 1-3, 1-4, 1-5, 2-0, 2-1, 2-2, 2-3, 2-4, 2-5
0-0, 0-1, 1-1, 1-2, 1-3, 2-3, 2-4, 2-5
scores that are at most one inning out-of-date: 2-3, 2-4, 2-5
after reading 1-3: 1-3, 1-4, 1-5, 2-3, 2-4, 2-5
for the writer: 2-5 for anyone other than the writer: 0-0, 0-1, 0-2, 0-3, 0-4, 0-5, 1-0, 1-1, 1-2, 1-3, 1-4, 1-5, 2-0, 2-1, 2-2, 2-3, 2-4, 2-5

source: Replicated Data Consistency Explained Through Baseball

Consistency model (aka consistency semantics)

- Contract between processes and the data store
  - If processes obey certain rules, data store will work correctly

- All models attempt to return the results of the last write for a read operation
  - Differ in how “last” write is determined/defined

Strict consistency is sometimes referred as strong consistency.

Consistency models: Data-centric

- Strict consistency
  - reads returns the most recent writes
- Linearizability
  - sequential consistent and time ordering
- Sequential consistency
  - ops executed in some sequential order
- Casual consistency
  - reads are seen in the same order
- Eventual consistency

Consistency models: Client-centric

- Monotonic reads
- Monotonic writes
- Read your writes
- Writes follow reads

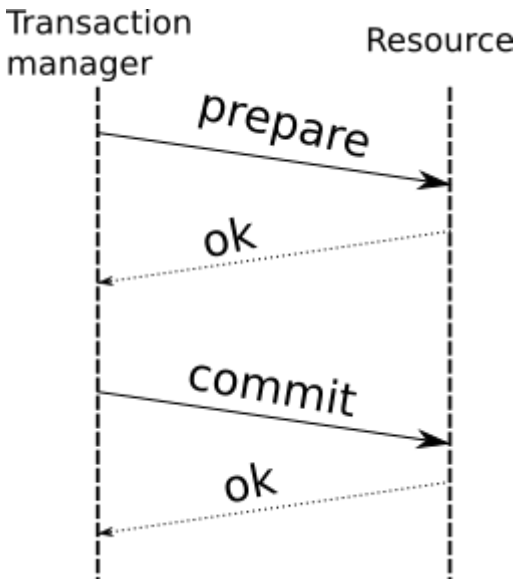
and some unordered notes…

- Strong consistency models (capable of maintaining a single copy)
  - Linearizable consistency: Under linearizable consistency, all operations appear to have executed atomically in an order that is consistent with the global real-time ordering of operations. (Herlihy & Wing, 1991)
  - Sequential consistency: Under sequential consistency, all operations appear to have executed atomically in some order that is consistent with the order seen at individual nodes and that is equal at all nodes. (Lamport, 1979)
    - Paxos. Paxos is one of the most important algorithms when writing strongly consistent partition tolerant replicated systems. It is used in many of Google’s systems, including the Chubby lock manager used by BigTable/Megastore, the Google File System as well as Spanner.
    - ZAB. ZAB - the Zookeeper Atomic Broadcast
    - Raft - easier Paxos
- Weak consistency models (not strong)

- Client-centric consistency models: many kinds of consistency models that are client-centric
- Causal consistency: strongest model available, strongest is global causal+ consistency – global as in needing to coordinate across datacenters, and the '+' to indicate that we care about convergence
- Eventual consistency models
  - Eventual consistency with probabilistic guarantees : Amazon's Dynamo (LinkedIn's Voldemort, Facebook's Cassandra and Basho's Riak based on that)
  - Eventual consistency with strong guarantees : CRDT, CALM

- http://lass.cs.umass.edu/~shenoy/courses/spring05/lectures/Lec15.pdf
- http://www.bailis.org/blog/understanding-weak-isolation-is-a-serious-problem
- https://www.microsoft.com/en-us/research/publication/replicated-data-consistency-explained-through-baseball

2PC



- CA (consistency + availability). Examples include full strict quorum protocols, such as two-phase commit.

2PC: atomic commitment protocol (ACP), a specialized type of consensus protocol

Why not 2PC (http://stackoverflow.com/questions/37297766/best-practices-of-distributed-transactionsjava)

- Some problems of 2PC comes from the fact that the coordinator is a single point of failure. If it is down then the system is unavailable, if there is a network partitioning and the coordinator happens to be in other partition than clients and resources then the system is also unavailable.
- Another problem of the algorithm is its **blocking nature**: once a resource has sent an agreement message to the coordinator, it will block until a commit or rollback is received. As a result the system can't use all the potential of the hardware it uses.

NOTE

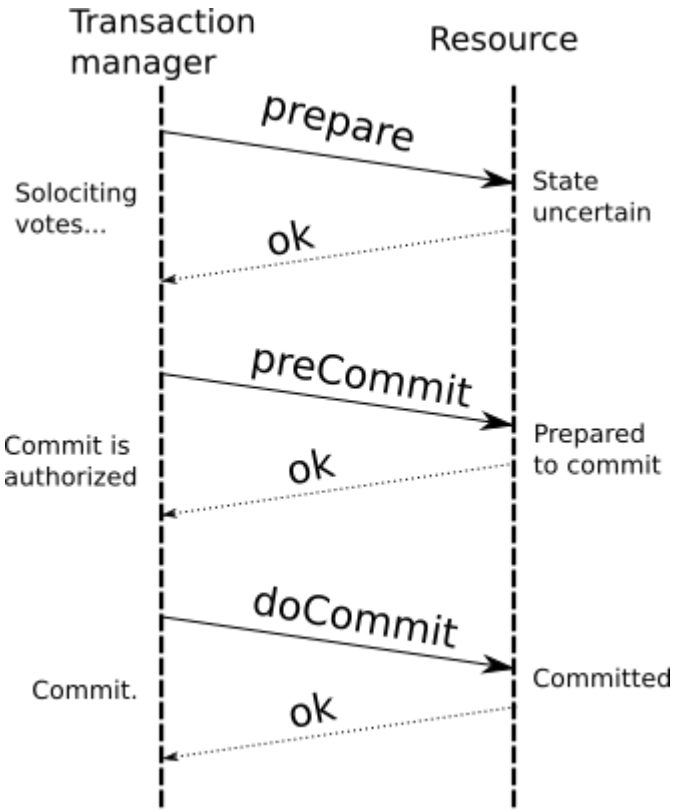
- it's blocking - it does not progress during the failure

In context of consesus: **2PC is safe but not live**

For 2PC can't be reach a safe decision when someone crashes/time-outs. Having only one participant down it means no-one can proceed.

- 2PC originated in year 1979 (Gray)
- 3PC in year 1981 (Stonebraker)
- Paxos in year 1998 (Lamport)
- http://the-paper-trail.org/blog/consensus-protocols-two-phase-commit
- http://highscalability.com/blog/2013/5/1/myth-eric-brewer-on-why-banks-are-base-not-acid-availability.html *

3PC



As it was known about 2PC that's safe but not live. Goal was creating consensus protocol to be live (never block on node failures).

3PC is live but it is not safe. 3PC has issue with network partitions where data correctness is not assured.

The protocol expects that node can't be just unreachable but living. When it stop to respond it's expected to be dead.

3PC splits commit/abort to two phases. * let to know to all participants what was the outcome * when all participants knows the outcome then commit (only that time)

NOTE

The base difference to 2PC is that in case of failure of participant or TM protocol defines further execution for the system. When TM is not available then after timeout participants can continue to work depending the phase they are in. After the TM or participant is back, recovery process restore the system.

- if one of the participants receives **preCommit**, they all can **commit**
- if none received **preCommit**, they all can **abort**
- https://roxanageambasu.github.io/ds-class//assets/lectures/lecture16.pdf
- http://planet.jboss.org/post/2pc_or_3pc
- http://the-paper-trail.org/blog/consensus-protocols-three-phase-commit
- https://cseweb.ucsd.edu/classes/wi17/cse291-d/applications/ln/lecture8.html

CAP - CP

http://thesecretlivesofdata.com/raft

- CP (consistency + partition tolerance). Basically those are **majority quorum protocols** in which minority partitions are unavailable such as Paxos, ZAB (**ZooKeeper**), Raft.

2PC, Paxos, and various approaches to quorum - these protocols provide the application programmer a façade of global serializability.

If you don't want to lose linearizability, you have to make sure you do all your reads and writes in one datacenter, which you may call the leader.

RAFT

Raft is a consensus algorithm for implementing fault tolerant distributed systems using a replicated state machine approach.

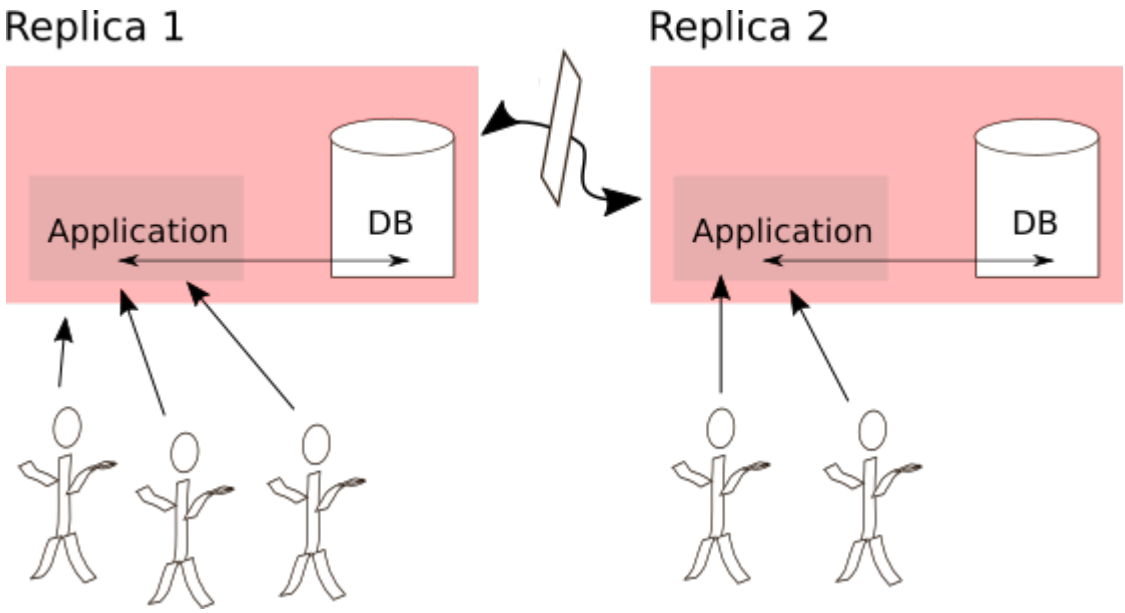
NOTE

Raft is "simplification" of Paxos, mainly for being easier to explain

There are two topics to talk about

- leader election
- log replication
- http://the-paper-trail.org/blog/distributed-systems-theory-for-the-distributed-systems-engineer
- http://thesecretlivesofdata.com/raft
- http://the-paper-trail.org/blog/consensus-protocols-paxos
- https://loonytek.com/2015/10/18/leader-election-and-log-replication-in-raft-part-1
- http://blog.nahurst.com/visual-guide-to-nosql-systems

CAP - AP



- AP (availability + partition tolerance). Basically those are **protocols using conflict resolution**, such as Dynamo.

NOTE

The easiest way how to resolve conflict is to move the responsibility to client. If there is a conflict DB returns to client both results and it's up to client what it does - uses both, rewrites one as the correct record...

- https://martin.kleppmann.com/2015/05/11/please-stop-calling-databases-cp-or-ap.html

CRDT, CALM

CRDT

conflict-free data type

CALM

consistency as logical monotonicity

Eventual consistency with strong guarantees : CRDT, CALM

CRDT

- a type of specially-designed data structure used to achieve strong eventual consistency
- basically they are datastructures that could be used with set of operation which are capable to resolve conflict. The basic example is a counter - it's implemented in way of not setting a state but list of additions. When split brain occurs then each partion is free to add addition (+1) to list of addition. When the cluster is joint again there is a function which defines how to resolve the conflict (e.g. part one receives request for +1 and part two three request for +1). The function just takes additions and sum that up. The resolution for the conflict is +4.

- Two types
  - Operation-based CRDTs
  - State-based CRDTs

CALM

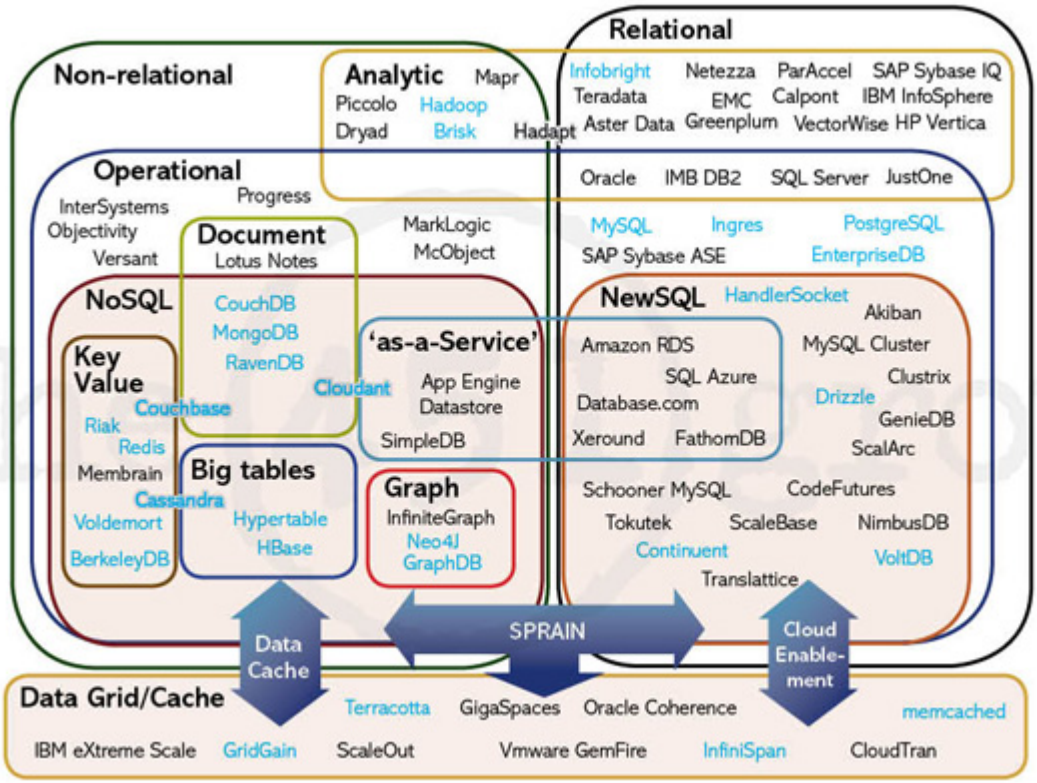
Moving responsibility to define where coordination is needed to creator of program. It provides tooling to define what are the "weak points" of the program and where coordination will be needed. Other parts are fine to be left as "AP" without any coordination enforced.

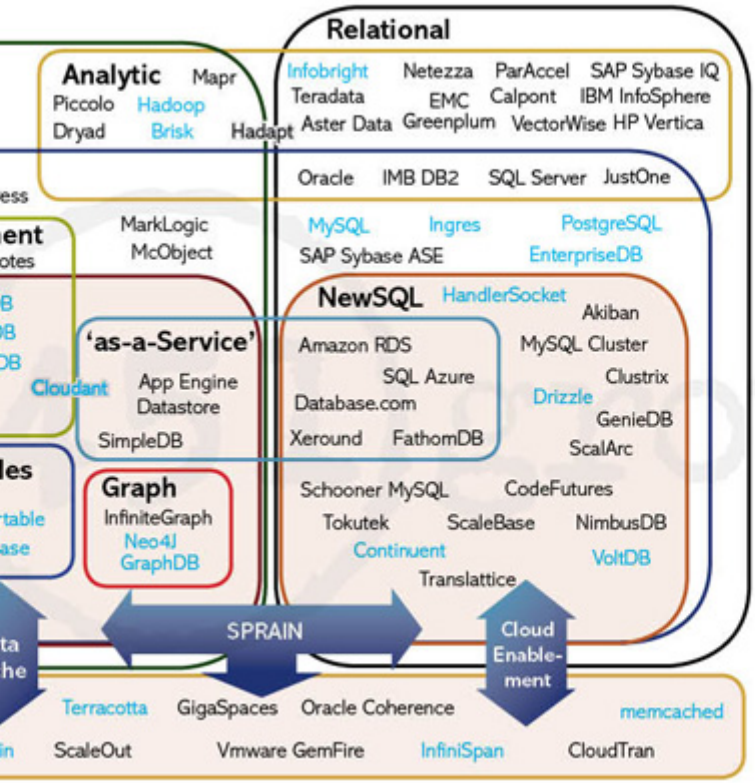
Accordingly, CALM tells programmers which operations and programs can guarantee safety when used in an eventually consistent system. Any code that fails CALM tests is a candidate for stronger coordination mechanisms.

A Bloom program may be viewed as a dataflow graph with external input interfaces as sources, external output interfaces as sinks, collections as internal nodes, and rules as edges. This graph represents the dependencies between the collections in a program and is generated automatically by the Bud interpreter.

- http://www.se-radio.net/2016/03/se-radio-episode-252-christopher-meiklejohn-on-crdts
- https://medium.com/@istanbul_techie/a-look-at-conflict-free-replicated-data-types-crdt-221a5f629e7e
- https://en.wikipedia.org/wiki/Conflict-free_replicated_data_type
- https://blog.acolyer.org/2015/03/16/consistency-analysis-in-bloom-a-calm-and-collected-approach
- http://www.bailis.org/blog/when-does-consistency-require-coordination
- http://queue.acm.org/detail.cfm?id=2462076
- https://blog.acolyer.org/2015/03/16/consistency-analysis-in-bloom-a-calm-and-collected-approach
- https://blog.acolyer.org/2015/09/08/out-of-the-fire-swamp-part-i-the-data-crisis
- https://blog.acolyer.org/2015/09/09/out-of-the-fire-swamp-part-ii-peering-into-the-mist
- https://blog.acolyer.org/2015/09/10/out-of-the-fire-swamp-part-iii-go-with-the-flow

SQL vs. NoSQL vs. NewSQL





- *source:* https://blogs.the451group.com

NOTE

Difference in terms of SQL, NoSQL and NewSQL is fuzzy. There is difference if SQL is used. But here in terms is rather kind of:

- **SQL** == **AC** (RBMS)
- **NoSQL** == **AP**
- **NewSQL** == **CP**

NewSQL examples NuoDB, VoltDB. Plus we can talk about systems like IBM HANA or possibly Google Spanner (when focused on strong consistency). And then probably even DynamoDB and CrockroachDB which added some stronger transaction abilities.

- https://blogs.the451group.com/information_management/2011/04/15/nosql-newsql-and-beyond
- http://dataconomy.com/2015/08/sql-vs-nosql-vs-newsql-finding-the-right-solution
- https://aphyr.com/posts/331-jepsen-voltdb-6-3
- https://www.nuodb.com/product/durable-distributed-cache
- http://www.methodsandtools.com/archive/acidnosql.html

Definition BASE

- **BA** for basic availability
- **S** for soft-state
- **E** for eventual consistency

- **Basic Availability** - The database appears to work most of the time.
- **Soft-state** - Stores don't have to be write-consistent, nor do different replicas have to be mutually consistent all the time.
- **Eventual consistency** - Stores exhibit consistency at some later point (e.g., lazily at read time).

BASE properties are much looser than ACID guarantees, but there isn't a direct one-for-one mapping between the two consistency models.

We can say that BASE transaction is used in NoSQL databases. As we can say that ACID transaction is used in SQL databases. We can say that ACID transaction is used in NewSQL databases.

NOTE

...nothing particularly correct in these saying but it's fine as simplification

BASE is a way how to get a distributed transaction (transaction over multiple resources/databases) being available.

- Technique known as 2PC (two-phase commit) for providing ACID guarantees across multiple database instances.
- ACID provides the consistency choice for partitioned databases, then how do you achieve availability instead? One answer is BASE.
- https://neo4j.com/blog/acid-vs-base-consistency-models-explained
- https://neo4j.com/blog/aggregate-stores-tour
- http://queue.acm.org/detail.cfm?id=1394128
- http://highscalability.com/blog/2013/5/1/myth-eric-brewer-on-why-banks-are-base-not-acid-availability.html

Distributed transactions

- Percolator's transactions
- RAMP transactions
- (Google Spanner)
- Compensating (SAGA) transactions

For Perclator and RAMP (and Spanner) We talk about transactions which is capable to work multiple records and provide atomic commit with them. For BASE we talk about eventual consistency where atomic commit is relaxed whole-over.

- If you want Serializable Isolation level then you should take a look on the Percolator's transactions. The Percolator's transactions are quite known in the industry and have been used in the Amazon's DynamoDB transaction library, in the CockroachDB database and in the Google's Pecolator system itself. A step-by-step visualization of the Percolator's transactions may help you to understand it.
- If you expect contention and can deal with Read Committed isolation level then RAMP transactions by Peter Bailis may suit you. I also created a step-by-step RAMP visualization.
- The third approach is to use compensating transactions also known as the saga pattern. It was described in the late 80s in the Sagas paper but became more actual with the raise of distributed systems.

source: StackOverflow how-to-manage-transactions-over-multiple-databases

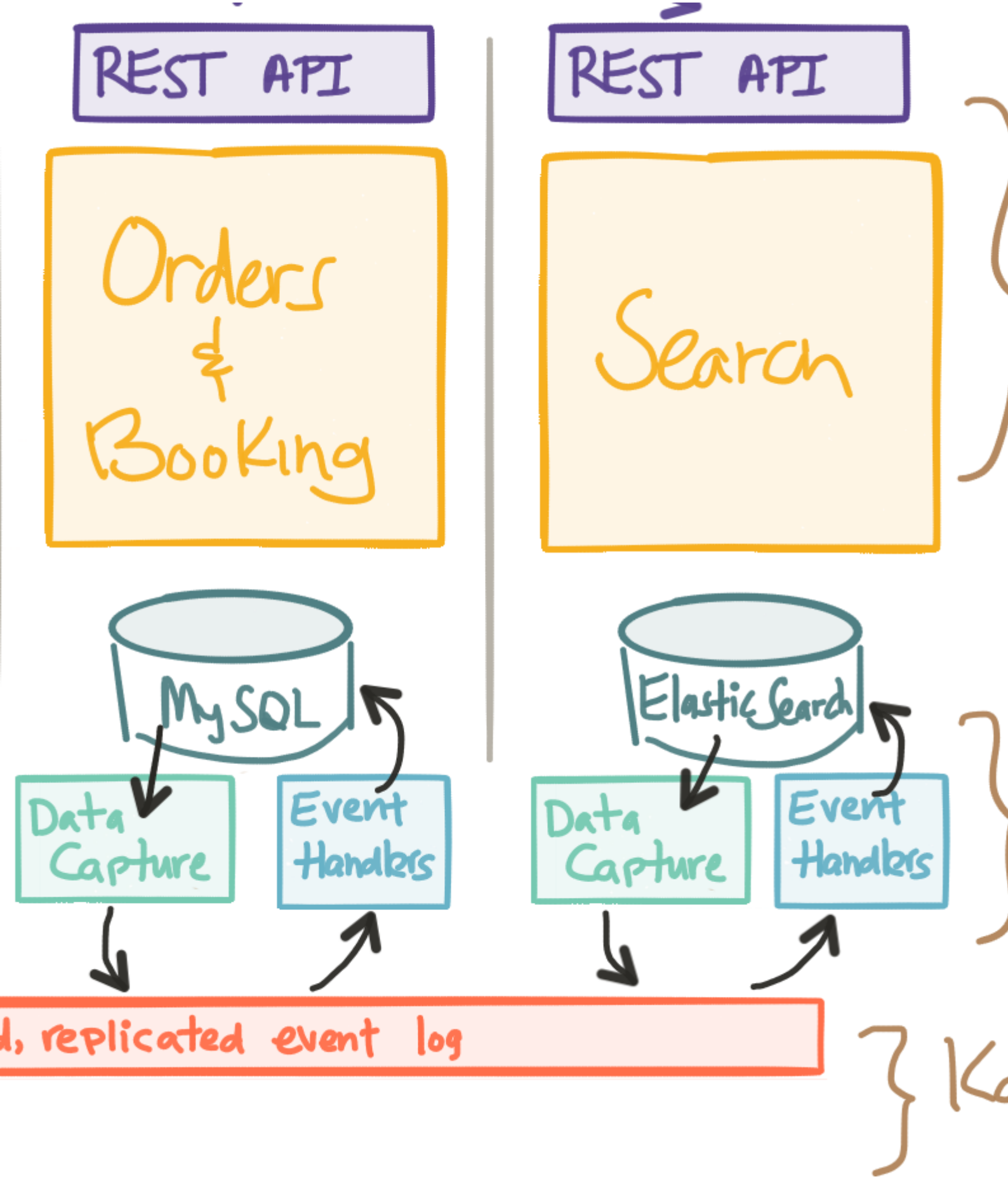
NOTE

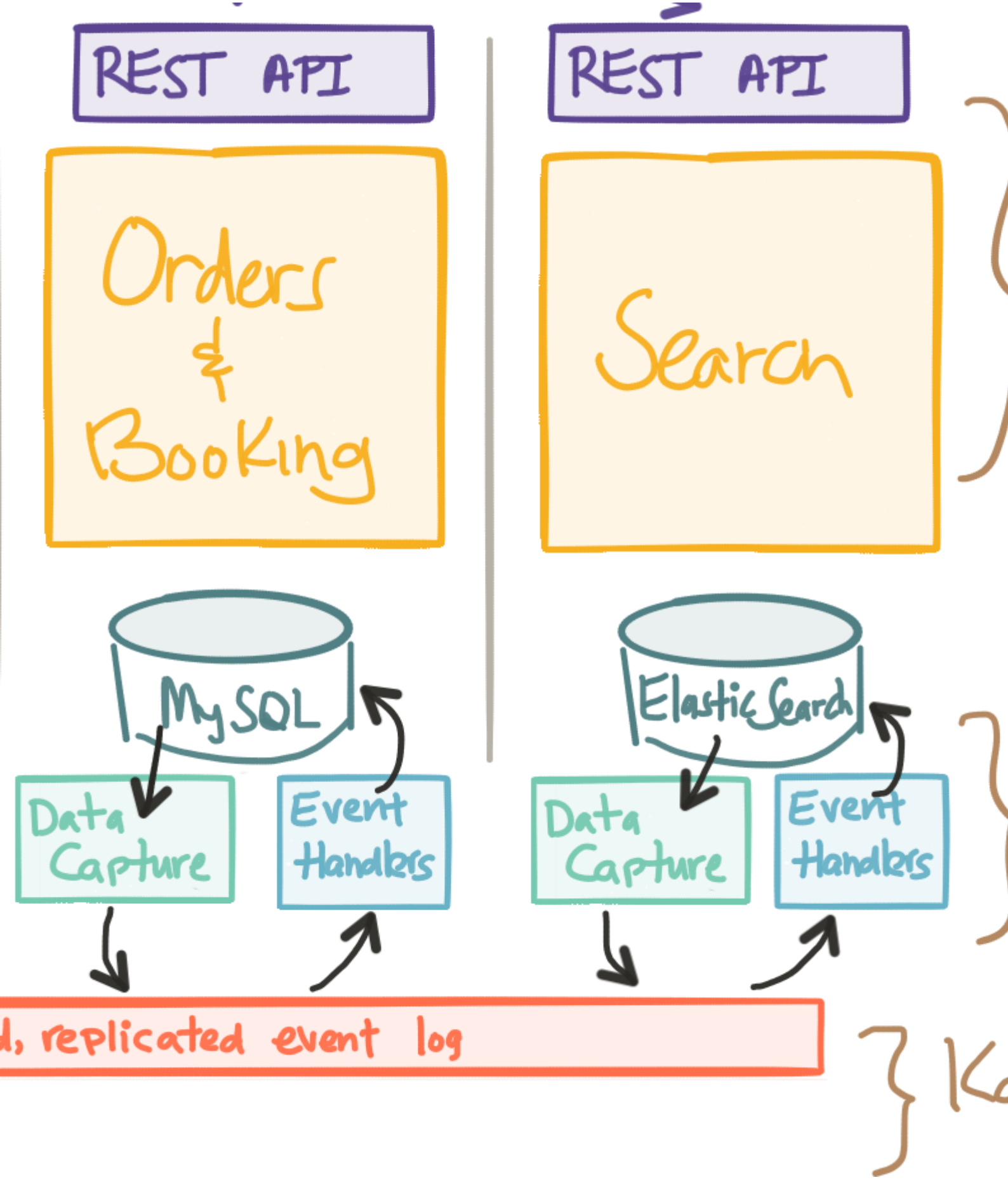
HAT for Highly Available Transactions * the paper comes with isolation/consistency analysis and introduces RAMP transactions

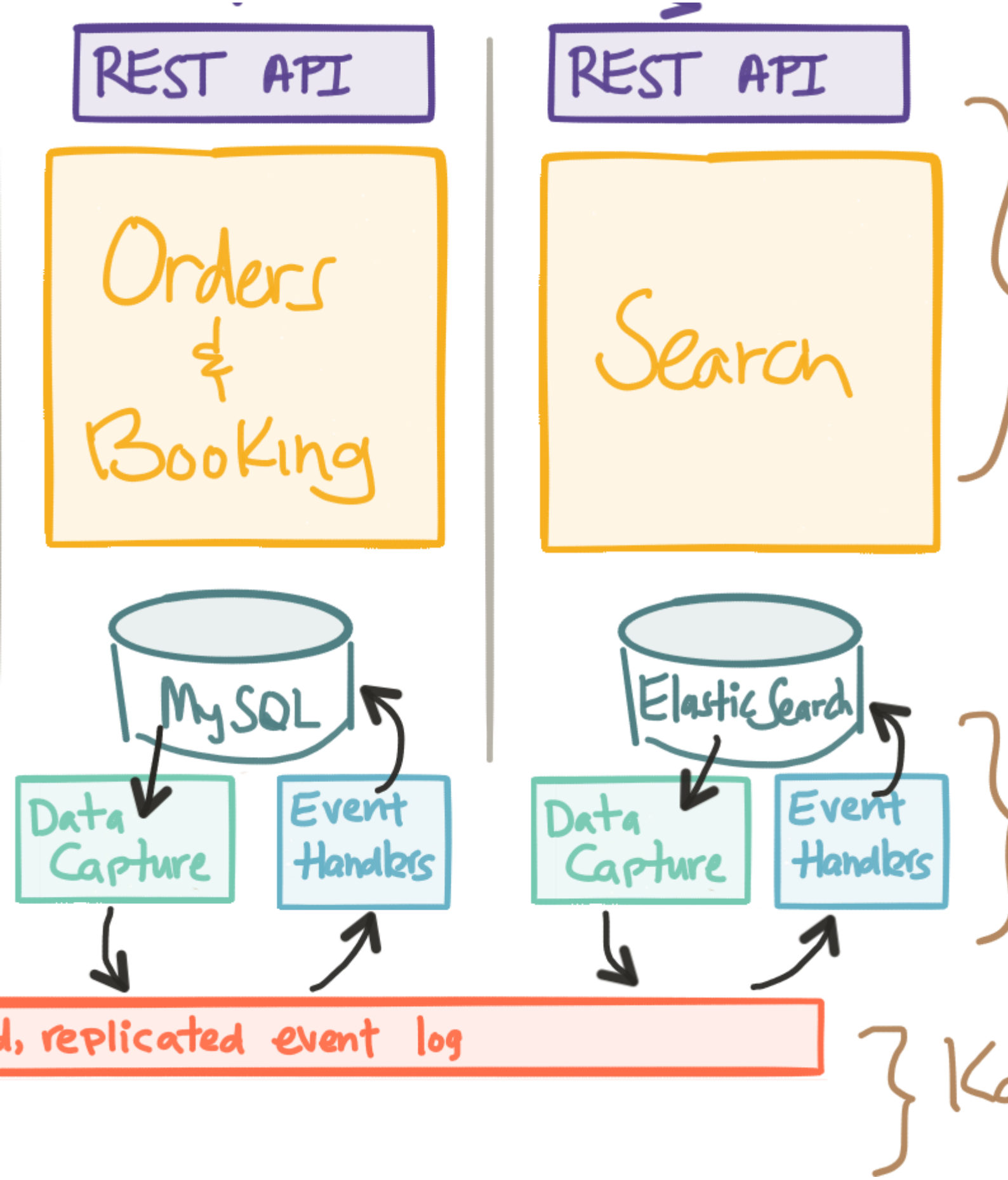
Google Spanner * It is a distributed relational database that can distribute and store data in Google's BigTable storage system in multiple data centers. Spanner meets ACID (of course, it supports transaction) and supports SQL. Currently, F1, Google's advertisement platform, uses Spanner. Gmail and Google Search will also use it soon. * Spanner is interesting by usage **TrueTime API**: TrueTime gets time information from GPS and the atomic clock

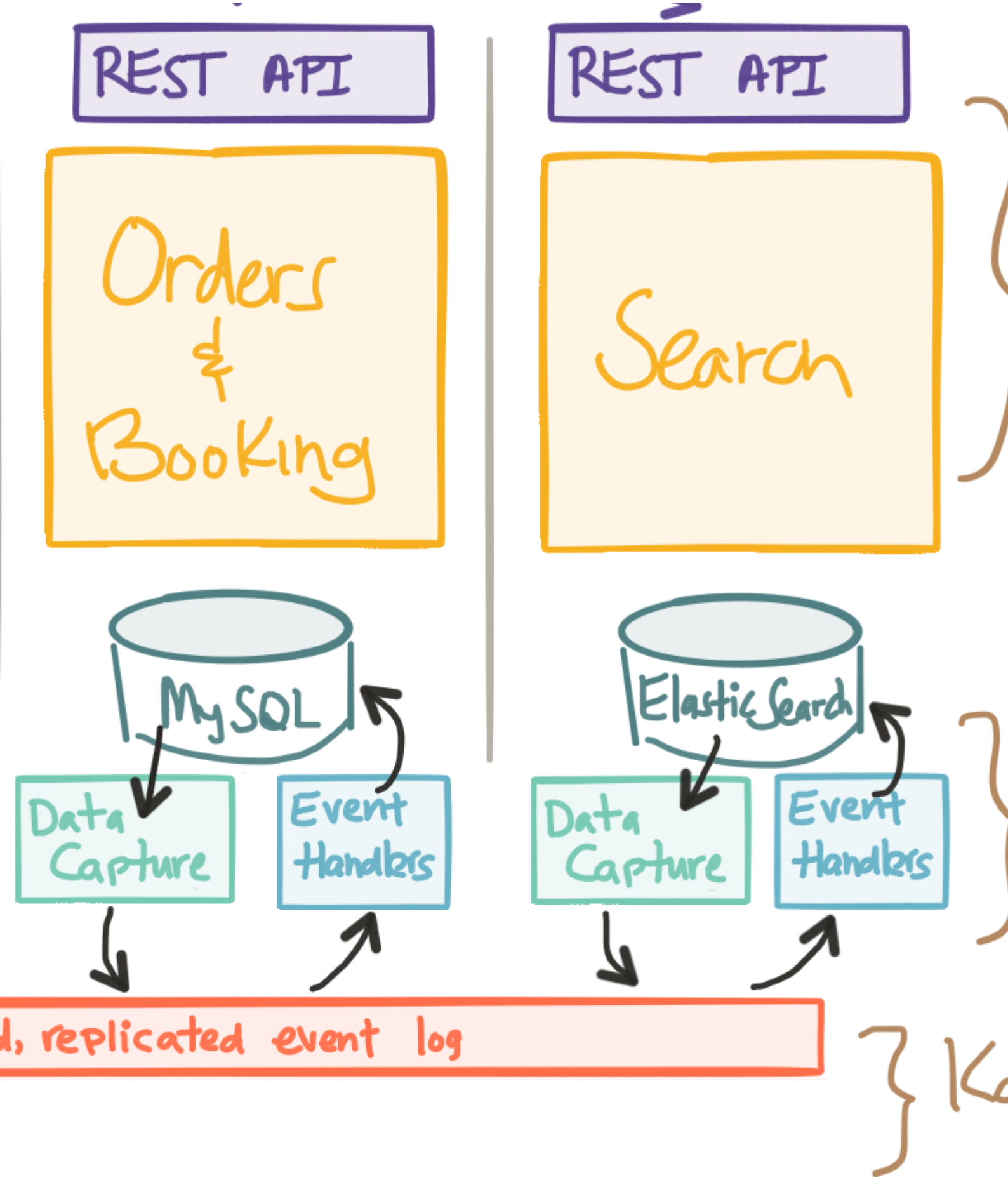
- http://www.bailis.org/blog/hat-not-cap-introducing-highly-available-transactions
- https://www.linkedin.com/pulse/client-side-transactions-distributed-data-stores-denis-rystsov
- https://www.youtube.com/watch?v=53DVkaW5Fb0
- https://www.youtube.com/watch?v=xDuwrtwYHu8
- http://www.bailis.org/blog/scalable-atomic-visibility-with-ramp-transactions
- https://dzone.com/articles/spanner-globally-distributed

MSA and consistency


REST API
REST API
Orders & Booking
Search
MySQL
ElasticSearch
Data Capture
Event Handlers
Data Capture
Event Handlers
d, replicated event log


REST API
REST API
Orders & Booking
Search
MySQL
ElasticSearch
Data Capture
Event Handlers
Data Capture
Event Handlers
d, replicated event log


REST API
REST API
Orders & Booking
Search
MySQL
ElasticSearch
Data Capture
Event Handlers
Data Capture
Event Handlers
d, replicated event log



source: The Hardest Part About Microservices: Your Data

In adopting the CQRS pattern for use in your application development, consider this transactional aspect of CQRS. Commands cannot be lost. You need a transaction manager (to handle ACID transactions) to ensure that every command is processed and that the events are generated and made persistent in the event store. This holds true for command handling, but if you consider the entire transaction (from running the command to the event listener execution) in regard to the asynchronous characteristics of flow, it is a BASE transaction.

Event Sourcing (ES) and **Command Query Responsibility Segregation** (CQRS) or **Turning the Database Upside Down**

from https://www.ibm.com/developerworks/cloud/library/cl-build-app-using-microservices-and-cqrs-trs

Implementation of CQRS saga base transaction is done in Axon Framework: http://www.axonframework.org

NOTE

- http://www.grahamlea.com/2016/08/distributed-transactions-microservices-icebergs : Why distributed transactions are bad in MSA
- http://blog.christianposta.com/microservices/the-hardest-part-about-microservices-data : Data management in MSA
- http://debezium.io : Red Hat to event sourcing for DBs
- https://kafemlejnek.tv/dil-6-nastupujici-architektury-web-aplikaci : Kafemlejnek.tv
- http://programio.havrlant.cz/kafka : Lukáš Havrlant blog
- https://github.com/cer/event-sourcing-examples : examples of http://eventuate.io
- https://www.infoq.com/articles/microservices-aggregates-events-cqrs-part-1-richardson : Developing Transactional Microservices Using Aggregates, Event Sourcing and CQRS - Part 1
- https://mapr.com/blog/how-stream-first-architecture-patterns-are-revolutionizing-healthcare-platforms
- https://dzone.com/articles/microservices-with-spring-boot-axon-cqrses-anddock

!

- Distributed systems: for fun and profit
- Design Data-intensive Applications
- Distributed systems theory for the distributed systems engineer

NOTE

Info dump from several sources, mixed, just for inspiration.

Scaling - vertical and horizontal

- Vertical scaling often creates vendor lock, further adding to costs.
- Horizontal scaling offers more flexibility but is also considerably more complex.

Partitioning - Partitioning is dividing the dataset into smaller distinct independent sets

- Replication improves performance by making additional computing power and bandwidth applicable to a new copy of the data
- Replication improves availability by creating additional copies of the data, increasing the number of nodes that need to fail before availability is sacrificed

Replication - Replication is making copies of the same data on multiple machines

- Replication improves performance by making additional computing power and bandwidth applicable to a new copy of the data
- Replication improves availability by creating additional copies of the data, increasing the number of nodes that need to fail before availability is sacrificed

Any horizontal scaling strategy is based on data partitioning; therefore, designers are forced to decide between consistency and availability.