**VICTORIA UNIVERSITY OF WELLINGTON**
***Te Whare Wananga o te Upoko o te Ika a Maui***

# *Data Versioning*

## *Lecturer :* *Dr. Pavle Mogin*

*SWEN 432*
*Advanced Database Design and Implementation*

## *Plan for Data Versioning*

- Influence of the replication on reads and writes
- Models of writing and reading as a function of the client consistency model
- Two phase commit,
- Master Slave model
- No Master model
  - Versioning
  - Quorum based two phase commit

  - ***Reedings:***
    - *Have a look at Useful Links on the Home Page*

## R / W Operations on Replicated Data

- Introducing replication into a partitioning scheme provides for the following benefits:
  - Enhanced reliability, and
  - Work load balancing
- But, replication also has a negative influence on read and write operations:
  - To accelerate writing, a client application may update only one replica, and then the update is subsequently asynchronously propagated to all other replicas in the background,
  - Different replicas of the same data object may be updated concurrently by different clients
    - As a consequence, different replicas may have different (even conflicting ) versions of the same data object

## A Motivating Example (1)

- Assume the following data object has been stored on two different replica nodes:

```
{“id” = 159, "name":“pavle",
"email":“pmogin@ecs.vuw.ac.nz"}
```

- Assume two client application simultaneously send the following updates to replica 1 and replica 2 under the eventual consistency level:

```
[client 1 to replica 1]: put(“id” = 159,
  "name":“pavle mogin", "email":
  "pmogin@ecs.vuw.ac.nz"})
[client 2 to replica 2] put(“id” = 159,
  "name":“pavle",
  "email":“pavle.mogin@windowslive.com"})
```

- Now, after concurrent updates, we have two conflicting versions of the same data object

## A Motivating Example (2)

- In this example:
  - The two concurrent writes changed the original data object value
  - But, there was no time available to propagate any of updates to the other replica before the other update has happen
  - So, we have got two conflicting versions of the same data object, and there is no rule which will tell the CDBS that it can throw away either the change to the `email` or the change to the `name` data or to retain the both
- So, we need either:
  - A model of operation that will not allow conflicting writes to happen, or
  - A versioning system that will allow detecting overwrites and throw away the old versions, and allow detecting conflicts and let the client reconcile these

## Client Consistency Models

- The partitioning and replication of data raise the following two questions:
  - How to dispatch a client request to a replica, and
  - How to propagate and apply updates to replicas
- The answers to the questions above depend on the client's consistency model:
  - Strict consistency,
  - Strong consistency,
  - Eventual consistency
  - There are also other consistency models, like:
    - Read your own writes consistency (RYOW),
    - Session consistency,
    - Monotonic read consistency

## ***Traditional Two Phase Commit (1)***

- The traditional two phase commit protocol provides for the **strict** consistency by bringing all replicas to the same state at every update
- Each update is coordinated by a node
- In the first (preparatory) phase, the coordinator sends updates to all replicas
  - Each replica writes data to a log file and, if successful, responds to the coordinator
- After gathering positive responses from all replicas, the coordinator initiates the second phase (commit)
  - Each replica writes another log entry and confirms the success to the coordinator

## *Traditional Two Phase Commit (2)*

- The traditional two phase commit protocol is not a popular choice for high throughput distributed database systems since it impairs availability
  - The coordinator has to wait quiet a number of network round trips and disk I/Os to complete
  - If a replica node crashes, the update is unsuccessful  (all or nothing behaviour)

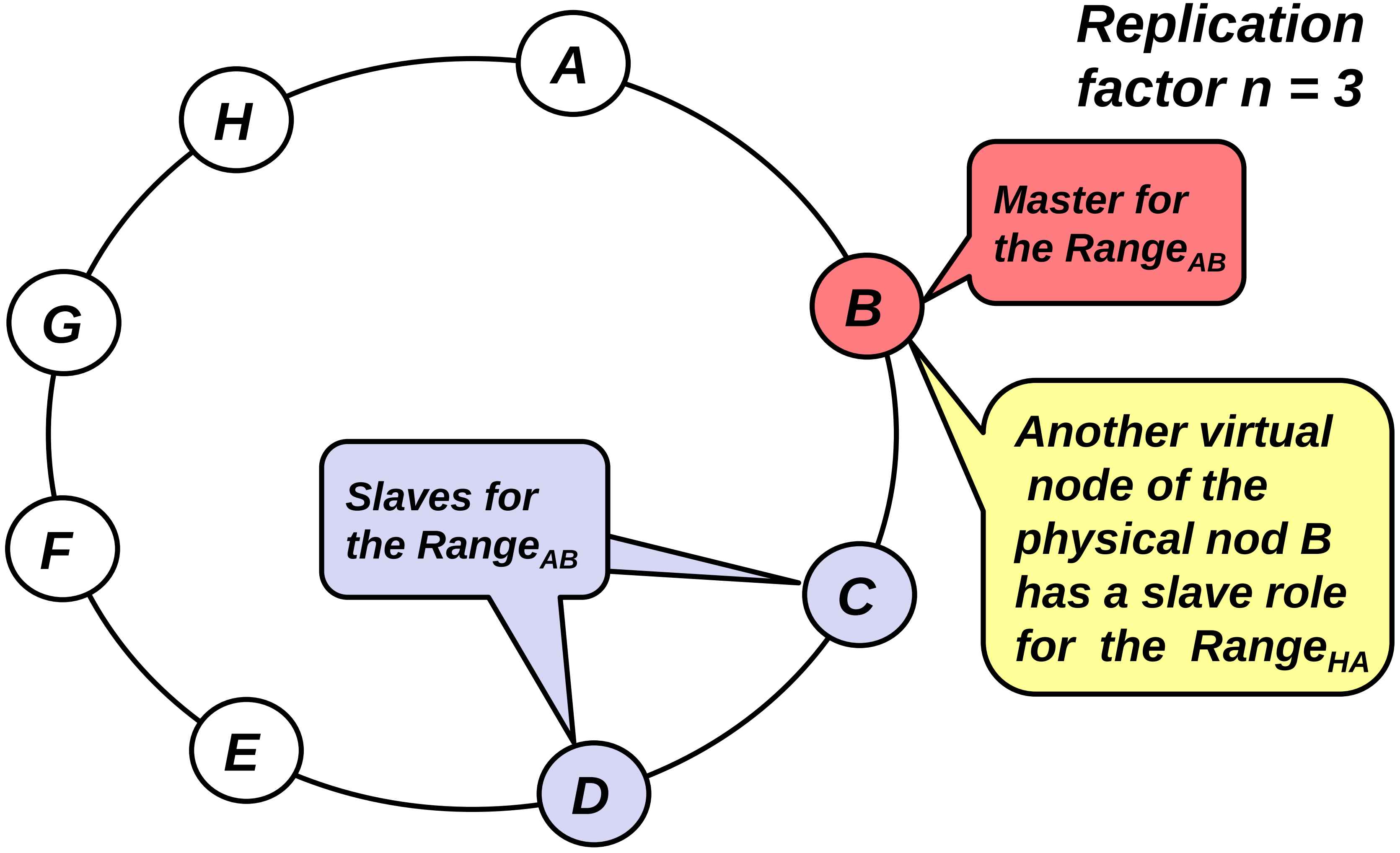
## ***Master Slave Model of Operation (1)***

- All data objects of a database partition are stored on a single master and multiple slave nodes
  - The master and slave nodes are virtual nodes of different physical nodes
- All ***update*** requests go to the master
  - The master updates the database partition and stores the update command in its log file on disk
  - Update commands are asynchronously propagated to slaves
  - The master is a SPOF, if it crashes before at least one slave is written, the update will be lost
  - The acknowledgement of the write to the client may be delayed until at least one slave has been updated
  - If the master crushes, the most  updated slave undertakes its role
    - But, until a new master has been elected, the database is unavailable for writes

## *Master Slave Model of Operation (2)*

- If the client can tolerate some degree of data staleness, ***read*** requests may go to any of replica
  - Good for load balancing
- If stale data are inacceptable, reads have to go to the master, as well
- Since all writes go to master, there is a potential for achieving isolation and atomicity
  - Transactional properties may be guaranteed at least for writes (and updates) involving a single database object
    - There are no transactional guaranties for updates of multiple objects within a single transaction
  - Offers a nearly strict consistency providing all masters are operational
- Since Master(s) are single point of truth, versioning is considered not to be needed

# MS Model with Consistent Hashing


Replication factor n = 3
A
H
G
F
E
D
C
B
Master for the Range$_{AB}$
Slaves for the Range$_{AB}$
Another virtual node of the physical nod B has a slave role for the Range$_{HA}$

## ***Master Slave Model of Operation (3)***

- A master may propagate updates to slaves in one of two ways:
  - State transfer and
  - Operation transfer
- In the ***state transfer***, the master sends its latest state to slaves, which replace their current with the received new state
  - It is robust, if a replica does not receive an update, it will be brought to a proper state by the next update
- In the ***operation transfer***, the master sends a sequence of update operations to slaves, which then execute them
  - Less traffic due to shorter messages, but the command order is crucial

## No Master Model of Operation

- The master slave model of operation works very well when:
  - There is a high read/write ratio, and
  - The updates are evenly distributed over the key range
- If any of the two conditions above is not satisfied, updates to any replica are allowed
- This model of operation requires a new technique for replica synchronization that is called data versioning

## *Data Versioning*

- If clients can tolerate a relaxed data consistency model:
  - Updates can go to any replica and then be propagated to others asynchronously by gossiping
  - This way of replication is called optimistic (also known as lazy replication), since replicas are allowed to diverge
  - Eventual consistency can be achieved by an anti-entropy protocol
  - The anti-entropy protocol performs replica converging
    - The anti-entropy protocols rely on data versioning
- ***Data versioning*** is achieved by appending such information to data objects that allows reasoning about the precedence of updates applied to object's versions
- There are two characteristic versioning mechanisms:
    - Timestamps and
    - Version Vector

## ***Timestamps***

- ***Timestamps*** are fields of time data type appended to each independently updatable data item
  - When a data item is updated, its timestamp field is set to the corresponding real (or logical) clock value
  - A CDBMS may keep several versions of the same data item, each with its value of the timestamp
  - The version of a data item having the greatest timestamp is considered as the actual value of the data item
- Timestamps seem to be an obvious solution for developing a chronological update order, but
  - Timestamps relay on synchronized clocks on all nodes
    - That may be a problem if replicas reside on nodes scattered around the world
  - If a CDBMS keeps only the last update of an item, timestamps do not capture causality (the update history) and do not allow conflict detection

## ***Lost Update with Timestamps***

**Eventual consistency level assumed**

| **Replica $R_1$** | **Replica $R_2$** | **Replica $R_3$** |
|---|---|---|
| Initially, x = [0, ts = 5] | Initially, x = [0, ts = 5] | Initially, x = [0, ts = 5] |
| $T_1$: x = x + 1 (ts = 6) | | $T_2$: x = x + 2 (ts = 7) |
| Send (x = [1, ts = 6]) → | Receive (x = [1, ts = 6]) | |
| | Since (ts = 5) < (ts = 6) | |
| | Set x = [1, ts = 6 | |
| | Send (x = [1, ts = 6]) → | Receive (x = [1, ts = 6]) |
| | | Since (ts = 6) < (ts = 7) |
| | | Retain x = [2, ts = 7] |
| | Receive (x = [2, ts = 7]) ← | Send (x = [2, ts = 7]) |
| | Since (ts = 6) < (ts = 7) | |
| | Set x = [2, ts = 7] | |
| Receive (x = [2, ts = 7]) ← | Send (x = [2, ts = 7]) | |
| Since (ts = 6) < (ts = 7) | | |
| Set x = [2, ts = 7] | | |

## *Version Vector (Informative)*

- The ***version vector*** is a mechanism for tracking changes to data in a distributed system where multiple clients might update data objects at different times
- The version vector allows determining whether an update to a data object preceded another update, followed it, or two updates happened concurrently
  - Concurrent updates may be conflicting
  - The detected conflicts have to be resolved by clients
- The version vector detects replication conflicts on a single data item, but not on two
  - This is achieved at the cost of additional data structures and complex update algorithms
  - Consequently slower than timestamps

## ***Quorum Based Commit Model***

- Quorum based commit model with data versioning is a more efficient way of providing consistency than the two phase commit protocol
  - It is also used with timestamps
- A coordinator issues an update request to all ***n*** replicas, but waits for a positive acknowledgement from only ***w*** (< ***n***) replicas
  - A read has to return response from ***r*** (< ***n***) replicas and CDBMS returns the ***latest object version*** to the client
  - After a read, all conflicting objects have to be reconciled in the background
  - The condition ***r*** + ***w*** > ***n***, guaranties a (relaxed) "*strict consistency*", called ***strong*** consistency

## ***Summary (1)***

- The replication has a negative impact on read and write operations:
  - Different replicas may have conflicting versions of the same data object
- Read and write modes of replicated data depend on client's consistency model ranging from strict to eventual consistency
- Master Slave mode of operation guaranties a (relaxed) strict consistency
- No master mode of operation relies on gossiping and uses one of, or combination of:
  - Data versioning and
  - Quorum based commit

## ***Summary (2)***

- Except in the case of quorum based commit, using gossip protocol even if there were no hardware outages or concurrent updates, unavoidably leads to the eventual consistency since the propagation of an update to all $n$ replica nodes requires time proportional to $\log_2 n$
- Two characteristic data versioning techniques are:
  - Timestamping and
  - Version vector
- The quorum based two phase commit also uses timestamp data versioning