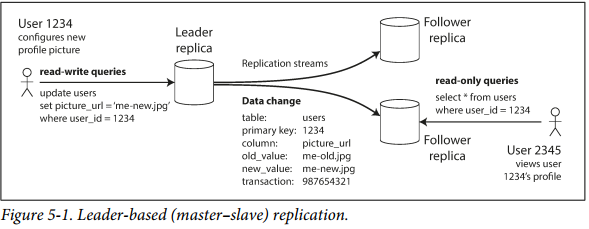
1. Leader-follower model:



* Synchronous replication đảm bảo cho follower có dữ liệu up-to-date với leader. Nếu leader fail, chúng ta vẫn có thể đảm bảo rằng trong follower vẫn có cùng dữ liệu với leader. Nhưng nếu follower fail, không respond lại, thì cả leader cũng vẫn bị block. Như vậy thực tế thì không ai config synchronous, vì chỉ cần 1 trong các node fail, thì cả hệ thống fail. Trong thực tế, nếu muốn config synchronous, thì chỉ cần config 1 node sync, các node còn lại async. Nếu như 1 node fail, slow, thì 1 node khác sẽ sync. Như vậy sẽ đảm bảo được luôn có 2 node up-to-date: leader và node sync đó. Đây được gọi là semi-synchronous.
* Async config có nhược điểm, nhưng thực tế thì được dung nhiều. Thực tế có nhều giải pháp để đảm bảo việc replicate vẫn high performed, high available. Đó là chain replication. Microsoft Azure Storage đã làm được điều này.
* Việc mở rộng thêm node có thể theo các bước sau:

1. Take snapshot của leader tại 1 thời điểm, cố gắng không lock database. Chức năng này được hỗ trợ trong backup.
2. Copy snapshot to follower node.
3. Bắt datachange từ điểm snapshot đó sang follower. PostgreSQL có log sequence number, MySQL có binlog coordinates.
4. Handling node outages:
5. Follower failure: Catch-up recovery

* Follower dựa vào log transaction để request leader stream datachange tiếp.

1. Leader fail:

* Một trong số các node phải trở thành leader, lúc này followers phải config lại để write những thay đổi sang leader mới. Quá trình này gọi là failover. Failover diễn ra các bước sau:

+ Xác định leader bị failed. Không có cách cụ thể để xác định leader bị fail hay không. Thông thường thì set timeout, khoảng 30s không phản hồi thì xem như leader dead.

+ Chọn leader mới. Leader mới thường là replica có dữ liệu up-to-date nhất.

+ Kết nối system lại với leader mới.

* Việc Failover cũng gặp một số vấn đề:

+ Khi leader mới chưa kịp nhận hết dữ liệu từ leader cũ, nhưng leader cũ đã fail rồi. Cách đơn giản là bỏ đi các dòng này.

+ Tuy nhiên bỏ đi các dòng này đôi khi gây ra lỗi nghiêm trọng. Ví dụ Github từng gặp trường hợp khi MySQL dùng autoincrement ID, nhưng leader cũ failed. Các ID này lại được tái sử dụng ở Redis. Điều này gây nhầm lẫn dữ liệu của user với nhau.

+ Split brain: trong 1 hệ thống có 2 node là leader. Cách dễ nhất là shutdown 1 node đi. Nhưng cẩn thận vì nếu không sẽ shut down cả 2.

1. Implementation of Repliacation Logs:
2. Statement-based replication:

* Lưu lịch sử statement.
* Tuy nhiên gặp trở ngại khi gặp các câu NOW(), RAND(),… hoặc các câu có dùng autoincrement ID, Trigger, Store,…

1. Write-ahead log (WAL) shipping

* Ghi lại hết log như data structure. PostgreSQL, Oracle có hỗ trợ điều này. Nhược điểm duy nhất là công việc này phụ thuộc hoàn toàn vào engine của database.

1. Logical (row-based) log replication

* Với 1 dòng thêm mới, log chứa đủ new values.
* Với 1 dòng được delete, log chứa đủ thông tin để xóa ID.
* VỚi 1 dòng update, log chứa đủ thông tin để update dòng đó.
* Đây gọi là kĩ thuật CDC (Change Data capture)

1. Trigger-based replication

* Sử dụng trigger và store.
* Databus của Oracle, Bucardo của Postgres.
* Cách này thì dễ dính bug và có nhiều giới hạn, nhưng được dùng nhiều vì độ linh hoạt của nó.

1. Các vấn đề của Replication Lag:
2. Read-after-write:

* Có 1 số cách để giải quyết Replication lag, một trong số đó là read-after-write:

+ Khi read những thứ đã được modified bởi chính user đó, thì read từ leader, còn lại read từ follower. Ví dụ profile thì thường được đọc, edit bởi chính user đó, nên đọc từ leader.

+ Nếu như mọi thử đều editable bởi user thì không thể làm vậy. Lúc này phải có criteria khác để quyết định có nên đọc từ leader hay không. Ví dụ track time của last update. Trong vòng 1 phút sau update thì đọc từ leader. Sau đó thì đọc từ follower.

+ Client biết được timestamp của lần write gần nhất, nên system có thể đảm bảo ít nhất cho đến timestamp đó.

* Một vấn đề nữa là cross-device read-after-write. Phát sinh 1 số vấn đề:

+ Không thể áp dụng kiểu nhớ timestamp, vì code ở các device khác không thể biết được timestamp update. Các metadata này phải được centralized.

+ Nếu replicas distributed datacenter. Thì phải đảm bảo rằng nó đọc được từ leader trên cùng 1 data center.

1. Monotonic Reads:

* Monotonic read: lesser than strong consistency, stronger than eventual consistency (việc đồng bộ nhưng không hẳn đảm bảo là có bao nhiêu replica bị fall behind, bao nhiêu dữ liệu bị trễ). Monotonic read đảm bảo rằng khi 1 user make several reads in sequence, thì họ sẽ không thấy time đi backward. Nghĩa là không thấy dữ liệu cũ nữa sau khi đã nhận được dữ liệu mới.
* 1 cách để đảm bảo điều này là đảm bảo rằng 1 user luôn đọc từ 1 replica duy nhất. Tuy nhiên nếu replica này fail, thì buộc phải chuyển sang replica khác.

1. Consistent Prefix Reads:
2. Multi-Leader Replication:

* Một số db có support multi-reader, 1 số tools như Tungsten Replicator MySQL, BDR PostgreSQL, GoldenGate for Oracle.
* Tuy vậy thì multileader có nhược điểm là conflict khi cùng 1 data nhưng được update cùng lúc ở 2 nơi khác nhau. Phải có cơ chế resolve conflict.
* Mặc dù vậy multi-leader ảnh hưởng đến nhiều công nghệ khác của db, vậy nên thường được avoided if possible.

1. Client có thể hoạt động offline:

* Một số ứng dụng có khả năng phải hoạt động được trong lúc offline. Lúc này phải có db local ở trong app, hoạt động như 1 leader, và có 1 cơ chế multi-leader replication async ở trong mọi device sử dụng app này.
* CouchDB được design cho việc này.

1. Collaborative editing
2. Handling Write Conflicts

* Write conflict occur khi có nhiều leader. Khi 1 nội dung nhưng được edit ở nhiều nơi.

1. Sync vs Async conflict detection:

* Đối với mô hình 1 leader, writer thứ 2 có thể block, đợi cho writer 1 complete, hoặc abort writer transaction thứ 2, bắt buộc user write lại. Mặt khác, ở mô hình multi-leader, cả 2 writes đều successful, và conflict có thể detected sau này. Vào lúc này thì có thể quá muộn để ask user resolve conflict.

1. Conflict avoidance:

* Cách dễ nhất để deal với conflict là cố gắng avoid them. Nếu như app đảm bảo rằng mọi write đều đi đến cùng 1 leader, thì conflict không thể occur.
* Ví dụ, trong trường hợp người user edit chính data của họ, thì app tự route đến chung 1 datacenter mà app đang read hoặc write. Lúc này giải pháp có thể là pick proxy của user và chọn leader.
* Tuy nhiên đôi khi vẫn phải đổi datacenter, khi datacenter fail. Lúc này avoidance sẽ fail.

1. Converging toward a consistent state:

* Một số cách để đạt được convergent giữa các replicas:

+ Đặt 1 unique ID (timestamp, long random number, UUID, hoặc hash), chọn ID cao nhất. Bỏ những writes còn lại. Nếu timestamp được dùng, người ta gọi đây là LWW (Last write wins). Cách này thì phổ biến, tuy nhiên dễ mất data.

+ Đặt 1 unique ID, ID cao hơn được ưu tiên.

+ Merge 2 values lại.

+ Record conflict lại và resolve sau.

1. Custom conflict resolution logic:

* Cách tốt nhất để xử lý conflict là tự viết custom conflict resolution trong ứng dụng multi-leader. Code này sẽ chạy mỗi khi write/read:

+ Write: Khi có conflict write, db sẽ gọi conflict handler. Bucardo cho phép viết conflict resolver bằng Perl.

+ Read: Khi xảy ra conflict, tất cả các conflict write sẽ được lưu lại. Ở lần tiếp theo read, db sẽ chạy resolver và write ngược lại db. CouchDB hoạt độgn theo cách này.

* Operational transformation: algorithm đằng sau Etherpad, Google Docs để xử lý conflict concurrency conflict.

1. Replication Topologies:

* MySQL support circular topology. Một số db khác hỗ trợ Star.
* Circular và star có vấn đề khi nếu 1 node chết, thì việc replication có thể gián đoạn. Việc reconfig thường thì tự làm manual. Ở đây thì rõ ràng all-to-all tốt hơn vì tránh được tình trạng single-point failure.
* All-to-all cũng có issue. Một số links có thể nhanh hơn các links khác. Kết quả là nhiều lúc overtake các link khác. Để tránh tình trạng này thì có 1 giải pháp là version vectors. Lưu ý rằng các hệ db nay cũng ít kiểm tra conflict.

1. Leaderless Replication:

* Riak, Cassandra, Voldemort sử dụng leaderless replication.

1. Detecting Concurrent Writes:

* Last write wins:

+ Old data sẽ bị overwritten and discarded.

+ Để xác định data nào tới trước, chúng ta có thể attach 1 timestamp cho mỗi write. Cassandra dùng cách này. Riak cũng có option để chọn cách này.

+ Cách tốt để dùng Cassandra là dùng UUID, mỗi khi được tạo ra thì key là unique.

* Có 2 loại write: “happens before” và “concurrent”
* Capturing happens-before:

+ Server giữ version number cho mỗi key, tang version number mỗi khi key được ghi, lưu version number với value được ghi.

+ Khi client read 1 key, server return values chưa bị overwriten, cũng như version number mới nhất. Client phải read key trước khi write.

+ Khi client write 1 key, thì phải include version number từ lần read trước, và phải merge all values từ lần read trước.

+ Khi server nhận 1 write với 1 version number bất kì, nó có thể overwrite tất cả values với version number đó, hoặc version number thấp hơn (vì các giá trị đã được merge thành new values rồi), nhưng phải keep all values với version number cao hơn nó (vì các values này concurrent với incoming write).

* Nếu sử dụng version number thì chúng ta có thể xác định trạng thái mà write phụ thuộc vào. Nếu không, thì write này sẽ không overwrite write nào cả, nó chỉ trả về 1 trong số các giá trị trong khi đọc liên tục.
* Version vectors: algorithm để keep track version number per replica. Riak sử dụng dotted version vector để xử lý điều này.