Big Data

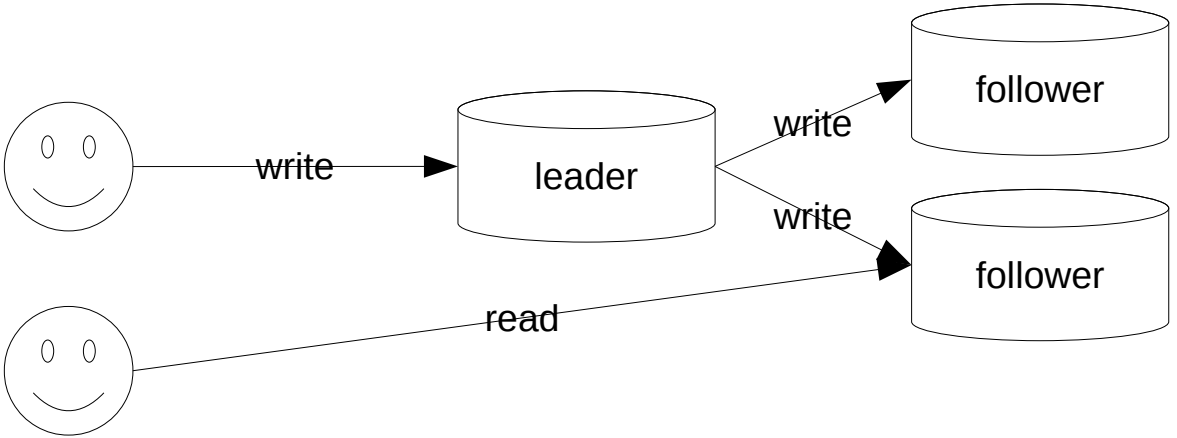
Replikation und Partitionierung

Andreas Scheibenpflug

Begriffe

- Replikation
  - Duplizieren der Daten von einem Knoten zu einem/mehreren anderen Knoten
  - → Ausfallsicherheit
  - → Skalierung von Lesezugriffen
- Partitionierung
  - Teilen der Datenmenge und Verteilen dieser auf mehrere Knoten
  - → Datenmenge zu groß für einen Knoten → Skalierung
  - → Beschleunigung von Lese- und Schreibzugriffen
  - Alias: s*hard* in relationalen DBs/MongoDB, *region* in Hbase, *vnode* in Cassandra/Riak

Replikation



- Wie lange blockiert write?
  - Synchrone vs. Asynchrone Replikation
- Was liest der 2. Benutzer?
- Wie werden Ausfälle von einzelnen Knoten behandelt?

Replikation - Arten

- Single Leader
  - Schreibzugriffe sind auf einen Knoten beschränkt
  - Von Followern darf nur gelesen werden
- Multi Leader
  - Schreibzugriffe sind auf mehrere Knoten möglich
  - Von Followern darf nur gelesen werden
- Leaderless
  - Alle Knoten sind gleichberechtigt
  - Lese- und Schreibzugriffe auf alle Knoten möglich

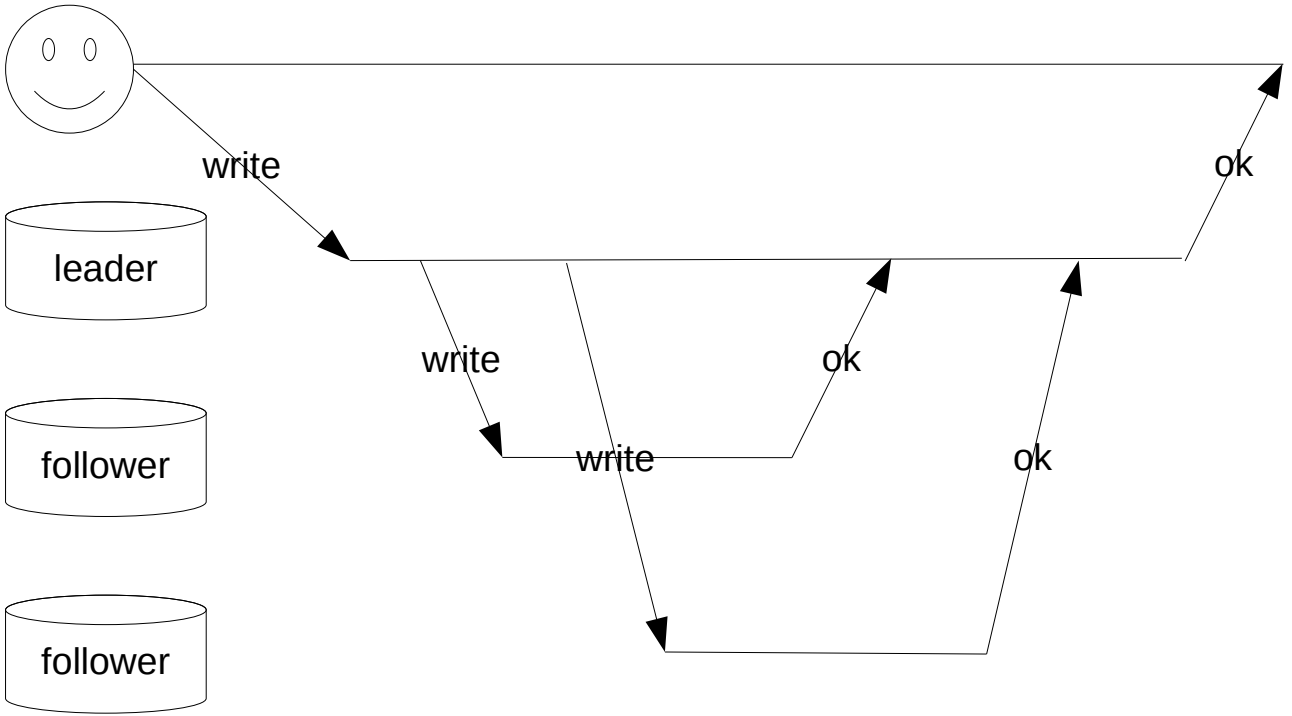
Single Leader

- Master/Slave System
  - PostgreSQL, MySQL, SQL Server, MongoDB, Kafka
- Lesezugriffe gehen an Leader und Follower
- Schreibzugriffe gehen nur an Leader
- Schreibzugriffe sind, je nach System, konfigurierbar oder nicht einstellbar
  - Synchron: Blockiert bis alle Follower Daten geschrieben haben (durable)
  - Asynchron: Kein Blockieren bis Daten auf allen Followern geschrieben sind (nicht durable)
  - Semi-synchron: Blockieren bis eine definierte Anzahl an Followern die Daten geschrieben haben

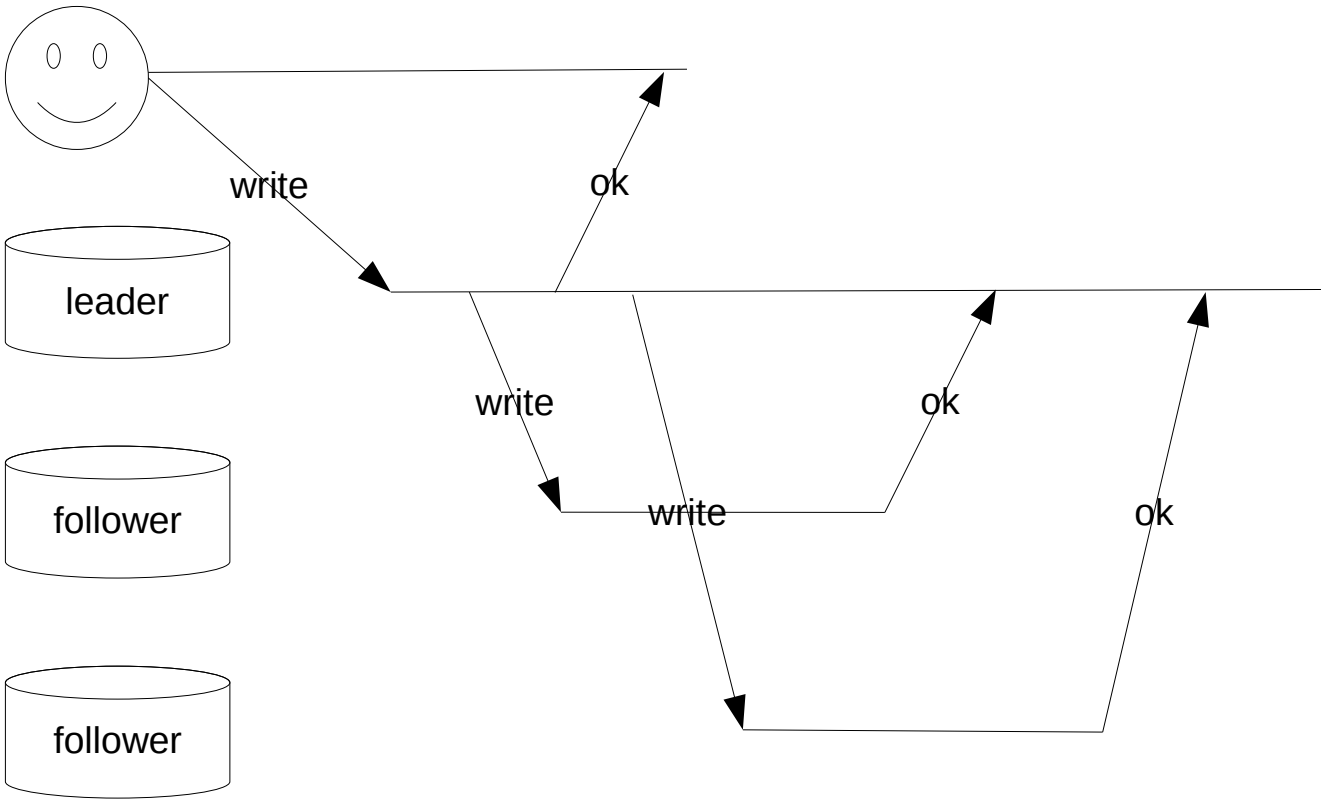
Asynchronität

- Vorteil: Performance
- Nachteile
  - Eventual Consistency: Lesen veralteter Daten
  - Durability: Bei einem Ausfall des Leaders können Daten verloren gehen
- Nachteil Synchronität
  - Performance
  - Fällt ein Follower aus, können keine Writes mehr durchgeführt werden
  - Daher wird oft eine semi-synchrone Konfiguration bevorzugt

Beispiel: Synchron



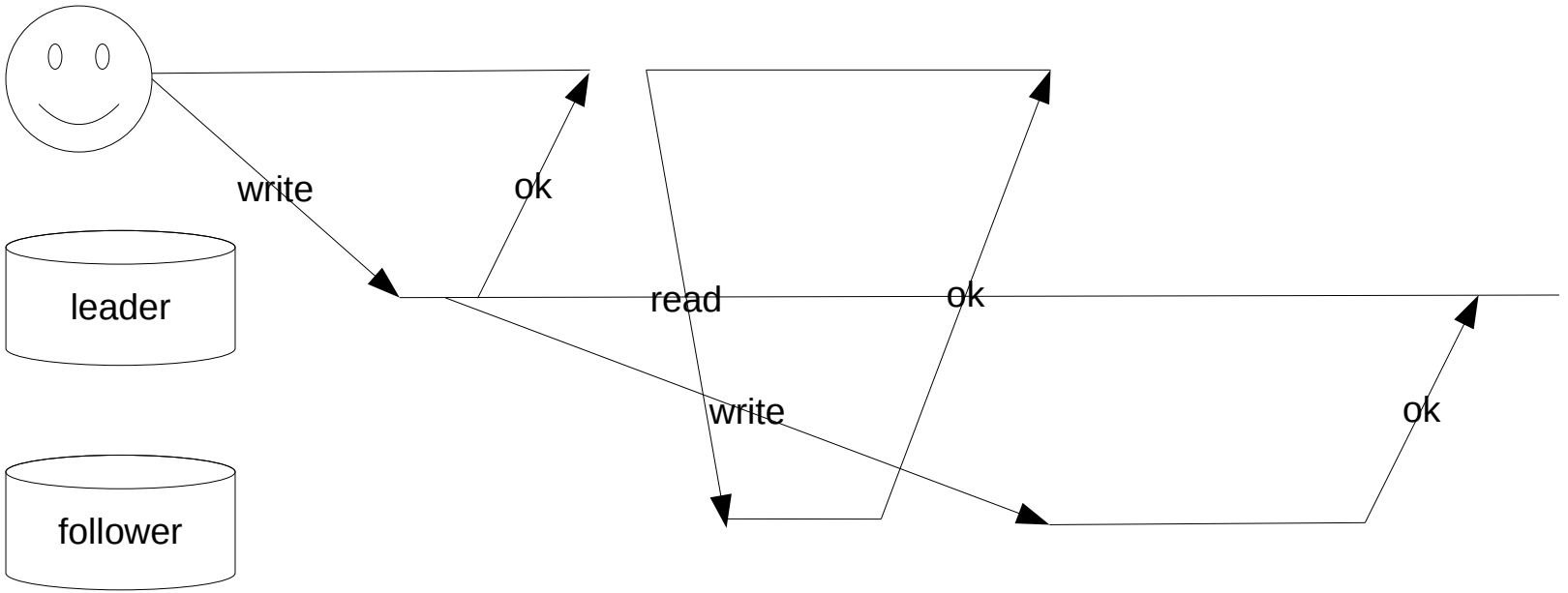
Beispiel: Asynchron



Ausfall von Knoten

- Follower
  - Keine Auswirkung
  - Ist der Follower wieder verfügbar, werden versäumten Writes vom Leader synchronisiert
- Leader
  - Promotion eines Followers zum neuen Leader
    - Election Process unter Followern (z.B. jener Follower mit dem aktuellsten Datenstand)
    - Controller: Im Vorhinein definierter Knoten
  - In asynchronen Systemen potentieller Datenverlust
  - Mögliche Probleme bei Wiedereingliederung des alten Leaders (z.B. Split Brain)

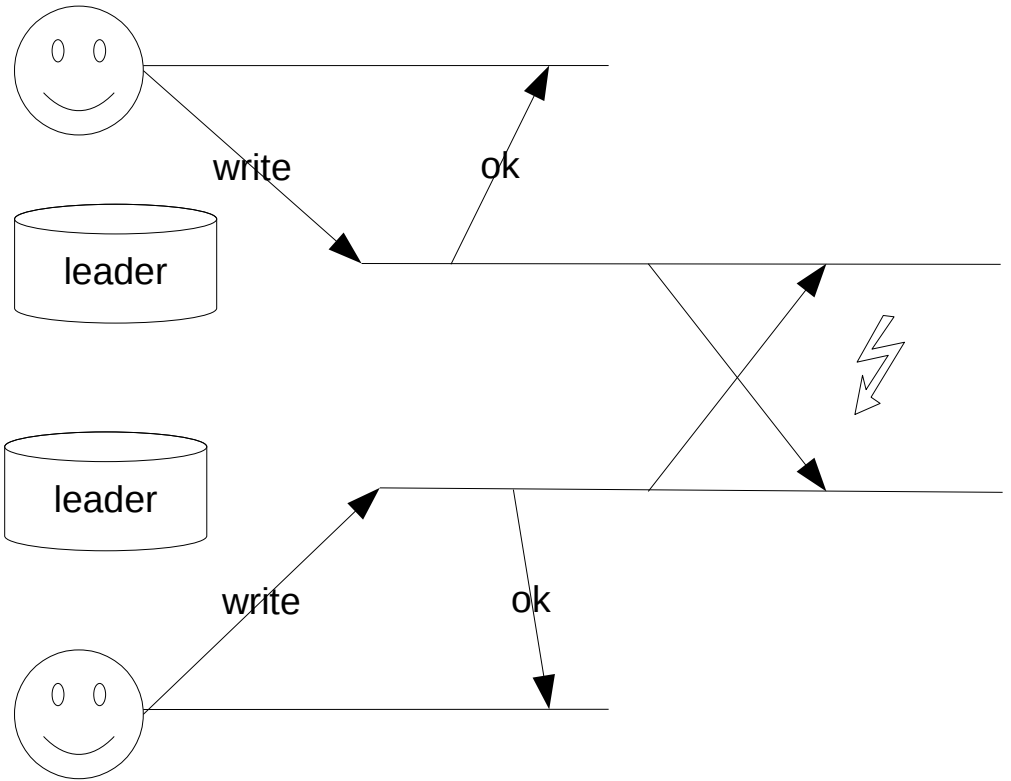
Replication Lag & Reading your own writes



Multi Leader

- Mehrere Leader die Writes akzeptieren
  - Bessere Performance
  - Ausfallsicherheit der Leader
- Leader müssen sich ähnlich den Followern synchronisieren
  - Im Normalfall asynchron
- Nachteil: Write Conflicts

Write Conflicts



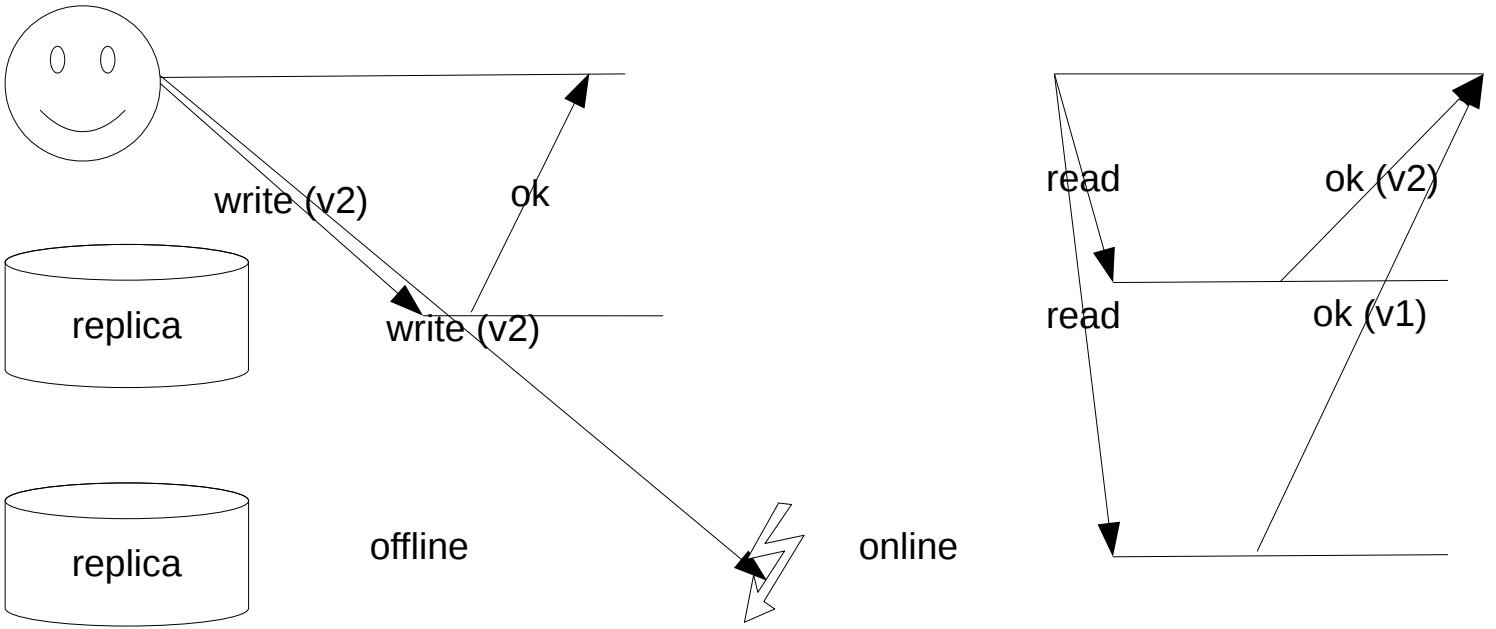
Conflict Resolution

- Vermeidung
  - Binden von Clients an einen Leader
- Last write wins
  - Jüngster Schreibvorgang wird behalten
  - Ältere Schreibvorgänge werden verworfen
- Anwendungslogik
  - Anwendung implementiert Code um Konflikte aufzulösen (z.B. Merge Operationen bei Wikis)
- Benutzer
  - Benutzer wird über Konflikt informiert und muss diesen manuell beheben (z.B. OneNote, Git)

Leaderless

- Lese- und Schreiboperationen sind auf alle Knoten des DB Systems möglich
- Knoten synchronisieren sich untereinander
- Beispiele: Dynamo, Cassandra, Riak
- Client schickt Requests entweder
  - an einen oder mehrere Knoten
  - an einen Koordinator der die Weiterleitung übernimmt

Ausfall von Knoten



Synchronisierung unter Knoten im Fehlerfall

- Read repair
  - Client liest von allen Knoten
  - Falls er alte Versionen der Daten erhält, schreibt er die neue Version auf die Knoten mit altem Datenstand
- Anti-entropy
  - Knoten vergleichen ihren Datenstand regelmäßig mit anderen Knoten
  - Werden veraltete Daten gefunden, werden diese aktualisiert

Quorums (1/2)

- *„Client schickt Requests an einen oder mehrere Clients“* → An wie viele?
- Quorum ist eine Möglichkeit zur Steuerung der Konsistenz unter Replikationen („Tuneable Consistency“)
- Parameter
  - *n*: Anzahl an Replikas (*n* != Anzahl an Knoten)
  - *w*: Anzahl an Replikas die einen Schreibvorgang als durchgeführt bestätigen müssen, um ihn als erfolgreich anzusehen → Request wird blockiert bis *w* Bestätigungen erhalten
  - *r*: Anzahl an Replikas von denen gelesen werden muss
- Diese Parameter sind in Dynamo-style DBs einstellbar
- Sinkt die Anzahl an Knoten (Ausfall) unter *w/r* → Operationen schlagen fehl

Quorums - Beispiele (2/2)

- *n=3, w=3, r=1*
  - Langsame Writes, Schnelle Reads, Konsistent
- *n=3, w=2, r=2*
  - Schnellere Writes, Langsamere Reads, Konsistent
- *n=3, w=1, r=3*
  - Schnelle Writes, Langsame Reads, Konsistent
- *n=3, w=1, r=1*
  - Schnelle Writes/Reads, Inkonsistent
- *w + r > n, w != n, r != n*
  - Konsistent und Ausfallssicher

Sloppy Quorums

- *n* Replikas sind fixen Knoten zugeordnet
- Wenn *w/r* von diesen *n* Replikas ausfallen, sind keine Schreib- und Leseoperationen mehr möglich
- Sloppy Quorum: Andere Knoten übernehmen temporär Writes/Reads
- Wenn Knoten wieder erreichbar sind, werden die zwischengespeicherten Writes weitergeleitet (hinted handoff)

Einige Probleme

- Concurrent Writes: Analog zu Conflict Resolution bei Multi Leader Systemen
- Sloppy Quorums: *n* kann während hinted handoff steigen → wenn *r/w* konsistent ausgelegt war, ist das möglicherweise während einem hinted handoff nicht mehr der Fall
- Bei manchen Systemen gibt es kein Rollback wenn ein Schreibvorgang nur teilweise erfolgreich war (<*w*)
- Wenn Abstand zwischen *w* und *n* groß ist, kann bei Knotenausfällen das Quorum während dem Schreiben von *n* - *w* verletzt werden
- → Konsistente Quorums sind keine absolute Garantie gegen Lesen veralteter Daten

Partitionierung

- Teilen der Datenmenge und möglichst gleichmäßige Verteilung auf mehrere Knoten
- Ein Datensatz soll möglichst effizient wieder gefunden werden
  - Zuordnung Datensatz → Knoten
- 2 Möglichkeiten
  - Partitioning by Key Range
  - Partitioning by Hash of Key

Partitioning by Key Range (1/2)

- Datensätze haben einen eindeutigen Schlüssel
- Es sind die Grenzen dieses Schlüssels bekannt oder können angenommen werden
  - Anzahl möglicher Schlüssel / Anzahl Knoten
- Beispiel Warenwirtschaftssystem
  - Artikelnummern: 1 – 10.000, 10 Knoten → 1.000 Artikel pro Knoten
  - Knoten 1: Artikel 1 – 1.000
  - Knoten 2: Artikel 1.001 – 2.000
  - Knoten 3: …

Partitioning by Key Range (2/2)

- Nachteile
  - Key Ranges können ungleichmäßig verteilt sein
  - Key Ranges können Hot Spots aufweisen
- Vorteile
  - Daten können sortiert gehalten werden
  - Range Queries: Effizientes Abfragen mehrerer Datensätze in einer Abfrage
- Bigtable, HBase, MongoDB < v 2.4

Partitioning by Hash of Key

- Berechnung des Hashes des Schlüssels
- Consistent Hashing: Hashfunktion liefert normalverteilte Ergebnisse
- Ermöglicht gleichmäßiges Verteilen von Datensätzen auf Knoten
- Nachteil: Range Queries sind nicht mehr (effizient) möglich
- Cassandra löst dieses Problem über Clustering Keys

Secondary Indexes

- By Document
  - Index wird für die Datensätze auf den Knoten für jeden Knoten erstellt
  - Suchen werden an jeden Knoten geschickt
  - Wird von den meisten DBs verwendet
- By Term
  - Index enthält Referenzen auf alle Datensätze über Knoten hinweg
  - Index wird partitioniert
  - Beschleunigt Suchen, Verlangsamt Writes
  - DynamoDB, Riak

Rebalancing

- Daten in Partitionen können wachsen oder schrumpfen
- Knoten werden zum System hinzugefügt oder entfernt
- Key Range: Dynamic Partitioning
- Hash of Key: Virtual Partitions

Dynamic Partitioning

- Key Ranges/Partitionen werden dynamisch an enthaltener Datenmenge angepasst
- Wenige Daten → Range wächst → Partition Merge mit benachbarter Partition
- Datenmenge wächst → Range schrumpft → Partition wird geteilt
- 1 Knoten kann mehrere Partitionen beherbergen
- MongoDB unterstützt Key Ranges und Hashes und führt für beide Dynamic Partitioning aus

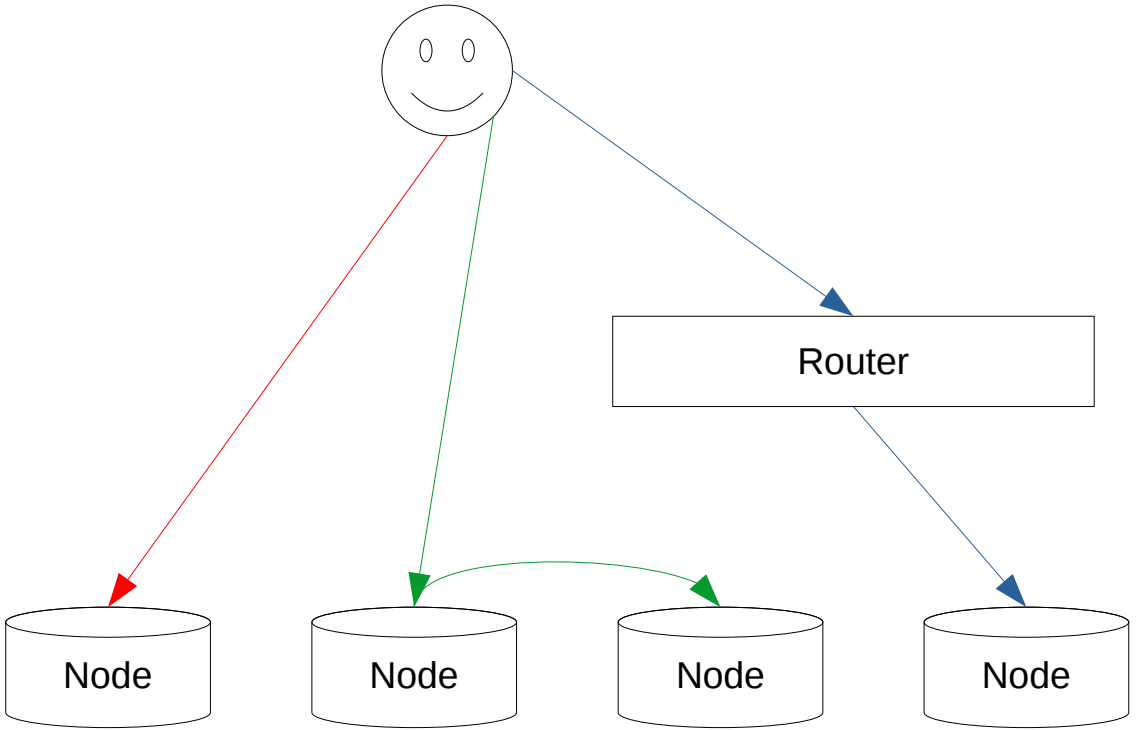
Node proportional partitions

- System hält eine fixe Anzahl an Partitionen
  - Je mehr Knoten → je weniger Partitionen pro Knoten
- Andere Möglichkeit: Fixe Anzahl an Partitionen pro Knoten
  - Je mehr Knoten → Aufteilung der Daten auf mehr Partitionen → Teilen von Partitionen

Request Routing (1/2)

- Wie kommt ein Client zu seinen Daten?
  - Kontaktieren eines Knotens, Weiterleitung
  - Seperater Routing Knoten
  - Client ist bekannt, auf welchem Knoten welche Daten liegen

Request Routing (2/2)



CAP Theorem

- Consistency: Jeder Benutzer sieht die selben Daten
- Availability: System ist zu jeder Zeit verfügbar und funktionstüchtig. Anfragen werden in akzeptabler Zeit beantwortet
- Partition tolerance: Bei Ausfällen ganzer Netzwerkpartitionen bleibt das System funktionstüchtig
- Es können nur max. 2 der 3 Eigenschaften im Fehlerfall erreicht werden
- Beispiele
  - Dynamo-style DBs sind meist AP Systeme
  - Relationale DBs sind meist CA Systeme