Mobile und Verteilte Datenbanksysteme

Roland Hediger

30. Juni 2014

Inhaltsverzeichnis

1 Einführung **3**
1.1 Fundamental Definitionen . . . 3
1.2 Verteilte Datenbank Definitionen . . . 3
1.2.1 Klassifizierung . . . 3
1.3 Dates 12 Regeln . . . 3
1.4 Aspekte Verteilte Datenbanksysteme . . . 4
1.5 Paralelle Datenbanksysteme . . . 4
1.5.1 Architekturen . . . 4
1.5.2 Aspekte . . . 4
1.6 Nosql Systeme . . . 4
1.6.1 Konzepte . . . 4
1.7 Mobile Datenbank Systeme . . . 5
1.8 Semantic Web . . . 5

2 Entwurf Verteilte Datenbanken **6**
2.1 Einfführung . . . 6
2.2 Entwürfe . . . 6
2.2.1 Top Down . . . 6
2.2.2 Bottom Up . . . 7
2.3 Korrektheit der Fragmentierung . . . 7
2.4 Fragmentierungsabkürzungen . . . 7
2.5 PHF . . . 8
2.6 PHF Schritt für Schritt . . . 8
2.7 PHF Beispiel Bikes . . . 8
2.8 DHF . . . 9

3 Vertikale Fragmentierung **10**
3.1 Zugriffseigenschaften . . . 10
3.2 Cluster Methode . . . 11
3.2.1 Äffinität . . . 11
3.2.2 Vertauschen bei Affinität . . . 11
3.3 Bond Energie Algorythmus . . . 11
3.4 VF - Vertical Fragmentation Example . . . 12
3.5 VF : Cluster möglichkeiten . . . 12
3.6 Schluss : VF . . . 12

4 Verteilte Anfrage Verarbeitung **14**
4.1 Query Processor . . . 14
4.2 Zentralisiert vs Dezentraliziert . . . 14
4.3 Überblick . . . 14
4.3.1 Ziel der Optimierung . . . 14
4.3.2 Komplexität . . . 14
4.4 Methode . . . 15
4.4.1 Verteilte Verarbeitung . . . 15
4.4.2 Zerlegung . . . 15
4.5 Lokalisierung . . . 15
4.5.1 PHF . . . 15
4.5.2 Schritten für Lokalisierung PHF . . . 15
4.5.3 Reduktion mit Join für PHF . . . 16
4.6 Reduktion für VF . . . 16

5 Concurrency **18**
5.1 Notation + Definitionen für Concurrency . . . 18

5.2 Einführung . . . 18
5.2.1 Aspekte der Transaktionsverarbeitung . . . 19
5.3 Arhitektur . . . 19
5.4 Nebenläufigkeit . . . 19
5.4.1 Beispiel : Global nicht serialisierbar . . . 20
5.5 Realisierung . . . 20
5.5.1 Zentrales 2PL . . . 20
5.5.2 Primary Copy 2pl . . . 20
5.5.3 Verteiltes 2PL . . . 20
5.6 Deadlocks . . . 21
5.6.1 Deadlock Erkennung . . . 21
5.6.2 Verteilte Deadlocks . . . 21

6 Verteilte Transaktionen 2 22
6.1 Lokale oder Verteilte Zuverlässigkeit . . . 22
6.2 Lokale Widerherstellung . . . 22
6.3 Zuverlässigkeit . . . 23
6.3.1 Komponenten . . . 23
6.4 2 Phase Commit . . . 23
6.4.1 Zustandsübergänge . . . 24
6.4.2 Recovery Protocols . . . 24
6.4.3 Ausfälle . . . 25
6.4.4 Probleme mit 2pc . . . 25
6.5 3 Phase Commit . . . 25

7 Replikation I 26
7.1 Einleitung . . . 26
7.1.1 Grunde für Replikation . . . 26
7.1.2 Ausführungsmodell . . . 26
7.2 Konsistenzmodelle . . . 26
7.3 Update Propogation Strategies . . . 27
7.4 Replikationsstrategieen . . . 27

8 Nosql 29
8.1 Einführung . . . 29
8.1.1 Motivation . . . 29
8.2 Cassandra . . . 30
8.3 Neo4J . . . 32
8.4 Bigtable . . . 33
8.5 MongoDB . . . 35

1 Einführung

Was wird Verteilt

- Aufbaulogik, Verarbeitungselemente
- Funktion
- Daten
- Steuerung

1.1 Fundamental Definitionen

Fundamental Principle To the user, a distributed system should look exactly like a nondistributed system.

Data Processing and Mobility It has become a common appproach to turn any location and situation a job of ce.

Warum Parallel Performance Performance And more Performance.

Big Data Volume,Variety,Velocity.

1.2 Verteilte Datenbank Definitionen

Verteilte Datenbank - DDB ine verteilte Datenbank ist eine Sammlung mehrerer, untereinander logisch zusammengehöriger Datenbanken, die über ein Computernetzwerk verteilt sind.

Verteiltes Datenbankverwaltungssystem (D-DBMS) Ein verteiltes Datenbankverwaltungssystem ist die Software, die die verteilte Datenbank verwaltet und gegenüber den Nutzern einen transparenten Zugang erbringt.

Verteiltes Datenbank System DDBS = DBS + D-DBMS

1.2.1 Klassifizierung

Heterogenität Hardware, Netzwerkprotokolle, Datenverwaltung, Datenmodell, Abfragesprache, Transaktionsverwaltung 2 Ausprägungen: homogen, heterogen.

Verteilung Verteilung: betrifft die Verteilung der Daten 2 Ausprägungen: verteilt, zentral

Autonomie Autonomie: betrifft die Verteilung der Steuerung 3 Ausprägungen: stark integriert, halbautonom, isoliert

1.3 Dates 12 Regeln

1. Lokale Autonomie
2. Unabhängigkeit von zentralen Systemfunktionen
3. Hohe Verfügbarkeit
4. Ortstransparenz
5. Fragmentierungstransparenz
6. Replikationstransparenz
7. Verteilte Anfragebearbeitung
8. Verteilte Transaktionsverarbeitung
9. Hardware Unabhängigkeit
10. Betriebssystem Unabhängigkeit
11. Netzwerkunabhängigkeit
12. Datenbanksystem Unabhängigkeit

1.4 Aspekte Verteilte Datenbanksysteme

Verteilter Datenbankentwurf

- wie die Datenbank verteilen
- Verteilung der DB mit Replikaten oder ohne
- Verzeichnisverwaltung

Anfragebearbeitung

- Zerlegung der Anfragen in ausführbare Instruktionen
- Anfrageoptimierung
- berücksichtigen von Verarbeitungs- und Datentransferkosten

Nebenläufigkeit

- Synchronisation konkurrierender Transaktionen
- Konsistenz und Isolation
- Deadlock Erkennung

Zuverlässigkeit

- Robustheit gegenüber Fehler
- Atomarität und Dauerhaftigkeit

1.5 Paralelle Datenbanksysteme

Parallele DB Systeme kombinieren Datenbankverwaltung und Parallel Verarbeitung zur Verbesserung der Performance und der Verfügbarkeit.

1.5.1 Architekturen

- Shared Memory Architecture
- Shared Disk Architecture
- Shared Nothing Architecture

1.5.2 Aspekte

- Parallele Anfrageverarbeitung
- Daten Partitionierung
- Parallelisierung von Operationen
- Lastausgleich
- Verfügbarkeit

1.6 Nosql Systeme

- Big Data
- Performance vs Scalability
- Latency vs Throughput
- Availability vs Consistency

1.6.1 Konzepte

- CAP Theorem [1]
- ACID vs BASE[2]
- Speicherstrukturen : Keyvalue, Document, Wide Column, Graph DB.
- Map / Reduce : Verarbeitung
- Consistent Hashing : Verteilung
- Multiversion Concurrency Control
- Paxos[3]

[1] Consistency, Availability , Partition Tolerance, will never achieve all three
[2] Basically Available, Soft State, Eventual Consistency
[3] Protokolle : Verfahren definiert - Griescisches Parlament lausig - wie kann man zu Entscheidung kommen

1.7 Mobile Datenbank Systeme

Mobile Datenbank Systeme sind Verteilte Datenbank Systeme mit zusätzlichen Eigenschaften und Einschränkungen:

- beschränkte Ressourcen
- häufig nicht verbunden
- verlangt andere Transaktions Modelle
- verlangt andere Replikationsstrategien
- Ortsabhängigkeit

1.8 Semantic Web

Die aktuelle Web Infrastruktur unterstützt ein verteiltes Ge echt von Webseiten, die gegenseitig mittels den sog. Uniform Resource Locators (URL) verknüpft sein können:

- Das Semantic Web unterstützt das Web auf der Ebene der Daten statt nur auf der Ebene der Darstellung
- Datenelemente können gegenseitig verknüpft sein, nicht nur Webseiten
- Datenelemente können gegenseitig verknüpft sein, nicht nur Webseiten
- Information über einzelne Entitäten können verteilt sein
- Daten Modell: Resource Description Framework (RDF)

$3^2 + 4x + 3x_{bla}$

2 Entwurf Verteilte Datenbanken

BIKES

BNr	BName	Preis	Typ	Bestand
B5	MCD03	4490.00	Road	2
B4	Siena	2390.00	Mountain	4
B2	City Cross	2190.00	Trekking	3
B3	Valiant	1090.00	Trekking	7
B1	Luxor	980.00	City	10
B6	Atlanta	890.00	Trekking	8
B7	Striker	890.00	Mountain	7

KUNDEN

KNr	KName	Ort
K1	Bike Outlet	Basel
K2	Swiss Bike	Olten
K3	Borer Velos	Basel
K4	City Bikes	Zürich
K5	MyBike	Aarau

AUFTRAEGE

ANr	Datum	KNr
A1	12.07.2009	K1
A2	01.09.2009	K2
A3	23.10.2009	K3
A4	05.11.2009	K4
A5	10.11.2009	K2
A6	19.11.2009	K1
A7	14.12.2009	K1
A8	17.12.2009	K2

APOSTEN

ANr	BNr	Menge
A1	B1	2
A2	B1	4
A2	B2	1
A3	B3	3
A3	B4	1
A4	B5	1
A5	B6	2
A6	B6	4
A7	B7	3
A8	B3	4

Abbildung 2.1: Beispiel DB

Anwendungen auf verschiedenen Knoten benötigen verschiedene Daten, unterschiedlich häufig:
A1: Liste der Aufträge mit günstigen Bikes (auf ORION in Basel)
A2: Änderungen an Bikes (auf CALYPSO in Aarau)
A3: Liste der Aufträge mit teuren Bikes vom Typ Mountain (auf TELESTO in Olten)
A4: Liste aller Kunden (auf ANANKE in Zürich)
Die Daten sollten dort abgelegt sein, wo sie am häu gsten gebraucht werden.

2.1 Einfführung

- Entscheid über die Platzierung von Daten auf den Knoten eines Computernetzwerks
- beeinflüsst die Performance der DDB und der Anwendungen
- lokaler Zugriff ist günstiger als Zugriff auf entfernte Knoten
- Analyse:

  welche Anwendungen (Queries)

  auf welchen Knoten

  benötigen welche Daten

  mit welcher Häu gkeit
- Resultat:

  Menge von Fragmenten (Ausschnitte der Daten)

  zugeteilt auf verschiedene Knote

2.2 Entwürfe

2.2.1 Top Down

beim Entwurf from scratch in homogenen Systemen nachgelagert an den konzeptionellen Entwurf.

1. wozu überhaupt fragmentieren?
2. wie fragmentieren?
3. wieviel fragmentieren?
4. wie prüfen der Korrektheit?
5. wie Fragmente zuteilen?
6. welche Informationen werden benötigt?

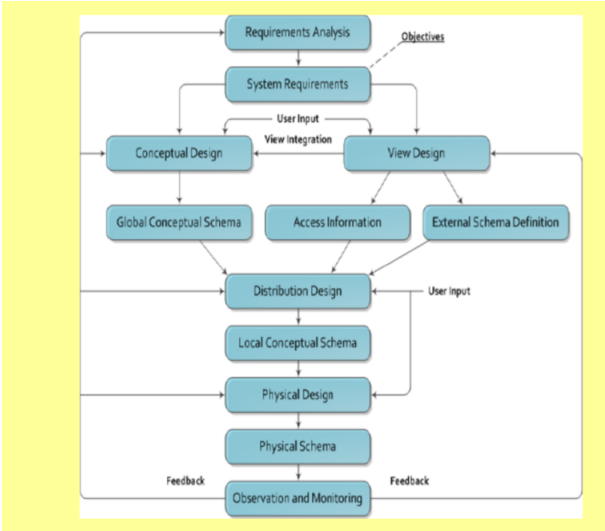

Abbildung 2.2: figure

Grad der Fragmentierung

- Vollständige Relationen sind zu grob, einzelne Attributswerte sind zu fein.

  geeignete Teile (Fragmente) der Relationen bestimmen

  Nebenläufikeit steigern bei mehreren Transaktionen mit Zugriff auf verschiedene Fragmente

  Zusatzkosten für Transaktionen mit Zugriff auf Fragmente an verschiedenen Knoten

Horizontal

Zerlegt aufgrund Riehen, mit Where klausel.

Projektion - Vertikal

Spaltenliste. In allen Fragmente PK muss dabei sein damit es rekonstruierbar ist- Kann kombiniert weden mit Horizontal für optimale Verteilung.

2.2.2 Bottom Up

- Multidatenbank Anwendungen
- Datenbanken schon auf verschieden Knoten vorhanden
- Problem der Datenintegration, Schemaintegration
- Web Services

2.3 Korrektheit der Fragmentierung

Vollständig Wenn R zerlegt wird $R1, R2$ $ldots R_n$ dann muss jedes Datenelement aus R in einem R_i enthalten sein. (Join mit gemeinsamen Pk - wiederzusammenstellung). (Horizontal umkehroperation - Union)

Rekonstruierbar Wenn wenn R zerlegt wird in , dann muss es $R1, R2$ $ldots R_n$ relationale Operatoren geben, so dass R wiederhergestellt werden kann.

Disjunkt wenn R horizontal zerlegt wird in $R1, R2$ $ldots R_n$, dann müssen die Fragmente paarweise disjunkt sein. wenn R vertikal zerlegt wird in $R1, R2$ $ldots R_n$, dann müssen die Fragmente bezogen auf die nichtprimen Attribute paarweise disjunkt sein.

2.4 Fragmentierungsabkürzungen

- Horizontale Fragmentierung (HF)

  Primäre horizontale Fragmentierung (PHF)

  Abgeleitete horizontale Fragmentierung (DHF) Fragmente gehören zu andere Fragmente wegen PK oder FK. Ähnliche Fragmentierung sinnvoll.

- Vertikale Fragmentierung (VF)
- Gemischte Fragmentierung (MF)

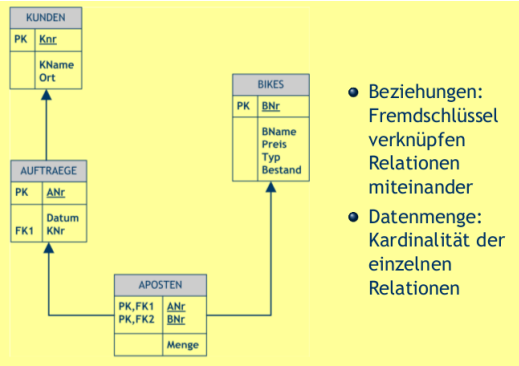

Abbildung 2.3: figure

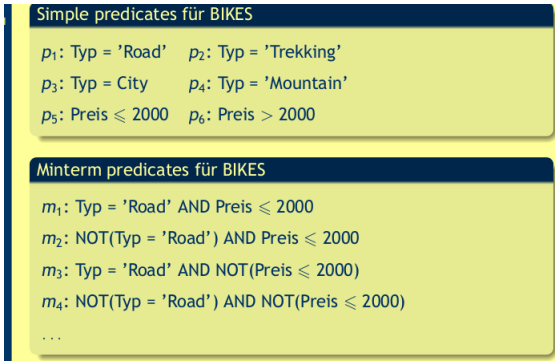

Abbildung 2.4: figure

2.5 PHF

Quantitive Informationen:
Bedingungen in Queries bestehen aus:

Simple Predicate p Vergleich eines Attributs mit einem Wert : Name = "bla" Preis $<$ 2000. Allgemein bestehenbedingungen aus boolschen Kombinierung von simple Predicates : $P = \{p_1, p_2 \ldots, p_m\}$ vom simple predicates bilden wir

Mintem predicate $M(P)$ **Verknupfungen aller simple predicates aus P mit AND und NOT**. bname= bla and not preis $<$ 2000 z.B
Bemerkung :
p_1 : bname = 'Sienna'
p_2 : preis $<$ 2000
$p_1 \wedge p_2$
$p_1 \wedge \neg p_2$
$\neg p_1 \wedge p_2$
$\neg p_1 \wedge \neg p_2$

Quantitive Informationen : Brauche ich das minterm oder nicht?

Minterm selectivity $sel(m_i)$ Anzahl Tuppel die mit dem Minterm m_i augewählt werden.

Access frequency $acc(m_i)$ Häufigkeit der Anwendungen auf Daten mit dem minterm m_i zugreifen

2.6 PHF Schritt für Schritt

Vollständigkeit Eine Menge von simple predicates ist vollständig genau dann, wenn auf beliebige 2 Tupel im gleichen Fragment von allen Anwendungen mit der gleichen Häu gkeit zugegriffen wird

Minimalität Wird durch ein simple predicate ein Fragment weiter aufgeteilt, dann muss es mindestens eine Anwendung geben, die auf diese Fragmente verschieden zugreift Ein simple predicate soll also relevant sein für die Bestimmung einer Fragmentierung Sind alle simple predicate eine Menge P relevant, dann ist P minimal

- Menge von Simple predicates mit der Eigenschaft der Vollständigkeit und Minimalität.

2.7 PHF Beispiel Bikes

Zugriffshäufigkeit gegeben je nach art des Fahrrads.
Anwendung 2 : Bikes aufgrund preise verwaltet.

Predicates aus beobachtungen
Anwendung 1: Zugriffshäufigkeit der Anwendung in excel, schauen. Wenn grosse verschieden - schlussfolgerung kann unterscheiden durch eine WHERE bedingung Anwendung 2 : Preis spielt hier eine rolle - ist relevante Bedingungen. Simple predicates aufnehmen die RELEVANT sind.

Verfahren erklären : Minimale Menge von simple predicates ausarbeiten aufgrund der Beobachtung Anwendungen. wahle eines aus von allen predicates das relevant ist - typ = mounten. erganze meine menge p' um die nächsten nur relevanten

sehe phf beispiel.
p1..p6 = P p1 ist relevant wege häufigkeit.
p2, p3 hinzufügen relevant
p4 nicht relevant weil : typ =city kann ich ausdrucken im minterm von anderen (Restmenge) p4 ist aber einfacher zu benutzen.
entweder p5, oder p6 aufnehmen sehe oben. MINIMALE MENGE. 2 HOCH 6 kombinationen. durch impl und wiedersprüche nur 8 sinnvoll.
Beispiel :
Typ = Road und Typ = Mountain ist ein wiederspruch. Bike für beides geht nicht.
Impl beispiel : Typ = road folgt nicht alle anderen typen. alle extrabedingungen mit nots reduzieren.
(Schritt 3) Fragmente bilden sehe folie.
homogen zugriff auf fragment ?? nutzung alle tupel gleich. zugriffhäufigkeit konform pro fragment.
wichtigsten anwendung betrachte 20 prozent 80 prozent last. kleinste sinnvolle einheit.

2.8 DHF

- ist eine horizontale Fragmentierung, die auf einer horizontalen Fragmentierung einer übergeordneten Relation basiert
- stellt sicher, dass Fragmente verschiedener Relationen, auf die häu g im Verbund zugegriffen wird, dem gleichen Knoten zugeteilt werden
- defnieren mit einem Semijoin (Dreickzeichen) [1]: Kann als normallen Join geschrieben werden : Projektion $\pi_{knr,datum,auftrage.knr}$ (Aufträge $\ltimes$ $Kunden_1$)

[1] Mit subquieries definiert: SELECT * FROM AUFTRÄGE WHERE EXISTS (SELECT 1 FROM $KUNDEN_1$ WHERE $KUNDEN_1$.KNR = AUFTRÄGE.KNR)

- wird Relation KUNDEN fragmentiert durch
  - $KUNDEN_1$: $\sigma_{Ort = 'Basel'}$ (KUNDEN)
  - $KUNDEN_2$: $\sigma_{Ort = 'Olten'}$ (KUNDEN)
  - $KUNDEN_3$: $\sigma_{Ort = 'Zürich'}$ (KUNDEN)
  - $KUNDEN_4$: $\sigma_{Ort = 'Aarau'}$ (KUNDEN)
- dann wird AUFTRAEGE fragmentiert mit
  - $AUFTRAEGE_1$: (AUFTRAEGE) $\triangleright$ ($KUNDEN_1$)
  - $AUFTRAEGE_2$: (AUFTRAEGE) $\triangleright$ ($KUNDEN_2$)
  - $AUFTRAEGE_3$: (AUFTRAEGE) $\triangleright$ ($KUNDEN_3$)
  - $AUFTRAEGE_4$: (AUFTRAEGE) $\triangleright$ ($KUNDEN_4$)

Abbildung 2.5: figure

3 Vertikale Fragmentierung

Geht um "ähnliche Nutzung".

- Vertikale Fragmentierung einer einzelnen Relation
- Modellieren des Zugriffs auf die Relationen

  Welche Anwendungen (Queries)

  verwenden welche Attribute
- und werden auf welchen Knoten
- wie häufig ausgeführt
- mit dem Ziel zu entscheiden

  welche Attribute werden häu g zusammen verwendet

  und sollen deshalb zum selben Fragment gehöre

BIKES

BNr	BName	Preis	Typ	Bestand
B5	MCD03	4490.00	Road	2
B4	Siena	2390.00	Mountain	4
B2	City Cross	2190.00	Trekking	3
B3	Valiant	1090.00	Trekking	7
B1	Luxor	980.00	City	10
B6	Atlanta	890.00	Trekking	8
B7	Striker	890.00	Mountain	7

KUNDEN

KNr	KName	Ort
K1	Bike Outlet	Basel
K2	Swiss Bike	Olten
K3	Borer Velos	Basel
K4	City Bikes	Zürich
K5	MyBike	Aarau

AUFTRAEGE

ANr	Datum	KNr
A1	12.07.2009	K1
A2	01.09.2009	K2
A3	23.10.2009	K3
A4	05.11.2009	K4
A5	10.11.2009	K2
A6	19.11.2009	K1
A7	14.12.2009	K1
A8	17.12.2009	K2

APOSTEN

ANr	BNr	Menge
A1	B1	2
A2	B1	4
A2	B2	1
A3	B3	3
A3	B4	1
A4	B5	1
A5	B6	2
A6	B6	4
A7	B7	3
A8	B3	4

Abbildung 3.1: figure

3.1 Zugriffseigenschaften

Für die Relation BIKES sind die Anwendungen als Queries gegeben:

q_1
```
SELECT bestand
FROM bikes
WHERE bname = ?
```

q_2
```
SELECT bestand, preis
FROM bikes
```

q_3
```
SELECT preis
FROM bikes
WHERE typ = ?
```

q_4
```
SELECT AVG(bestand)
FROM bikes
WHERE typ = ?
```

Abbildung 3.2: figure

Die Verwendung (Usage) der Attribute durch diese Anwendungen (Queries) wird in einer Matrix dargestellt: Matrix U. eine 1 bedeutet: Query verwendet Attribut eine 0 bedeutet: Query verwendet Attribut nicht

	BName	**Preis**	**Typ**	**Bestand**
q_1	1	0	0	1
q_2	0	1	0	1
q_3	0	1	1	0
q_4	0	0	1	1

Anwendung wird ebenfalls in einer Matrix festgehalten: Matrix Acc.

	S1	**S2**	**S3**
q_1	15	20	10
q_2	5	0	0
q_3	25	25	25
q_4 3	0	0	

die Zahl 15 oben links besagt z.B.: Query q_1 wird auf Knoten S1 15 Mal pro Tag ausgeführt

3.2 Cluster Methode

- Der Zugriff auf jede Attributsmenge durch die Anwendungen ist sehr unterschiedlich

  Dadurch würde jedes Attribut sein eigenes Fragment bilden, was wenig sinnvoll ist

  Besser wird nach Attributsmengen gesucht, auf die ähnlich zugegriffen wird

- Die Clustering Methode erkennt ähnliche Zugriffsmuster auf Attribute

  Für jedes Paar von Attributen bestimmen, wie oft auf sie gemeinsam zugegriffen wird

  Cluster bilden von Attributen mit hoher Af nität

3.2.1 Äffinität

$$aff(A_i, A_j) = \sum_{k:U_{k_i}=1, U_{k_j}=1} \sum_{l=1}^{s} acc_{k_l} \tag{3.1}$$

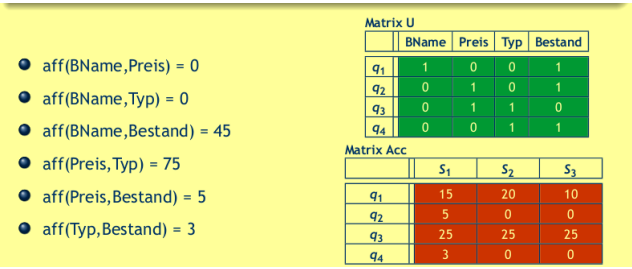

Abbildung 3.3: figure

Matrix AA Zeigt Äffinität zweier Attribute :

	BName	**Preis**	**Typ**	**Bestand**
BName	45	0	0	45
Preis	0	80	75	5
Typ	0	75	78	3
Bestand	45	5	3	53

Skalarprodukt benachbarter Spalten misst die Ähnlichkeit des Zugriffsmusters: Af nität der Nachbarschaft (bond):

$$\sum_{z=1}^{n} = aff(A_z, A_x) * aff(A_z, A_y) \tag{3.2}$$

Die Summe über alle Skalarprodukte ist die globale Af nität der Nachbarschaft

BName, Preis: 45 x 5 = 225
Preis, Typ:
80 x 75 + 75 x 78 + 5 x 3 = 11865
Typ, Bestand: 75 x 5 + 78 x 3 + 3 x 53 = 768

globale Affinität: 12858

Vertauschen ist auch möglich:

3.2.2 Vertauschen bei Affinität

Durch Austausch von Spalten (und den entsprechenden Zeilen) in der Matrix AA verändert sich die globale Af nität der Nachbarschaft:

	BName	**Preis**	**Typ**	**Bestand**
BName	45	0	0	45
Preis	0	80	75	5
Typ	0	75	78	3
Bestand	45	5	3	53

Preis, BName: 5 x 45 = 225
BName, Typ: 45 x 3 = 135
Typ, Bestand:
75 x 5 + 78 x 3 + 3 x 53 = 768
Globale Affinität: 1128

3.3 Bond Energie Algorythmus

- Gegeben $n \times n$ Matrix AA der Affinitäten

  Beliebige 2 Spalte aus AA wählen und in Resultats Matrix CA stellen

- Iteration:

  eine der übrigen $n - i$ Spalten so in Resultats Matrix positionieren ($i + 1$ mögliche Positionen), dass sich der grösste Beitrag an die globale Affinität der Nachbarschaft ergibt

  Die Zeilen entsprechend den Spalten anordnen

- Beitrag einer Spalte A_k wenn zwischen A_i und A_j platziert:
  $\text{cont}(A_i, A_k, A_j) = \text{bond}(A_i, A_k) + \text{bond}(A_k, A_j) - bond(A_i, A_j)$

3.4 VF - Vertical Fragmentation Example

Matrix AA

	BName	Preis	Typ	Bestand
BName	45	0	0	45
Preis	0	80	75	5
Typ	0	75	78	3
Bestand	45	5	3	53

Abbildung 3.4: Gegebene Matrix der AA Affinitäten

Matrix CA

	BName	Typ		
BName	45	0		
Preis	0	75		
Typ	0	78		
Bestand	45	3		

Abbildung 3.5: Aus der Affinitäts Matrix AA werden die Spalten BName und Typ beliebig gewählt

Spalte Preis an 1. Position platzieren und Beitrag an globaler Affinität berechnen:
cont(_, Preis, BName) = 0 + 225 - 0 = 225
Spalte Preis an 2. Position platzieren und Beitrag an globaler Af nität berechnen:
cont(BName, Preis, Typ) = 225 + 11865 - 135 = 11955
Spalte Preis an 3. Position platzieren und Beitrag an globaler Af nität berechnen:
cont(Typ, Preis, _) = 11865 + 0 - 0 = 11865

Preis an der 2. Position bietet den grössten Beitrag.

- Mit der Spalte *Bestand* ebenso verfahren
- Die 2. Position für *Bestand* gibt den grössten Beitrag
- Anordnen der Zeilen entsprechend den Spalten

In der Resultats Matrix *CA* sind die Clusters erkennbar für eine optimale vertikale Fragmentierung
Es sind aber noch 2 weiter Clusterungen möglich
Wie kann die optimale berechnet werden?

Matrix CA

	BName	Bestand	Preis	Typ
BName	45	45	0	0
Bestand	45	53	5	3
Preis	0	5	80	75
Typ	0	3	75	78

Matrix CA

	BName	Bestand	Preis	Typ
BName	45	45	0	0
Bestand	45	53	5	3
Preis	0	5	80	75
Typ	0	3	75	78

Abbildung 3.6: figure

3.5 VF : Cluster möglichketen

- Trennpunkt kann entlang der Diagonalen verschoben werden
- Für jede der Möglichkeiten berechnen, wie oft:

  ausschliesslich auf Attribute eines Fragments zugegriffen wird (gut)

  auf Attribute beider Fragmente zugegriffen wird (schlecht)

- Trennqualität $sq = acc(VF_1) \times acc(VF_2) - acc(VF_1, VF_2)^2$ maximieren

  acc(VF 1): Summe Zugriffshäu gkeiten der Queries ausschliesslich auf VF 1

  acc(VF 2): Summe Zugriffshäu gkeiten der Queries ausschliesslich auf VF 2

  acc(VF 1 , VF 2): Summe Zugriffshäu gkeiten der Queries auf VF 1 und VF 2

3.6 Schluss : VF

Aus der erhaltenen Clusterung der Attribute, ergänzt mit dem Primärschlüssel ergeben sich die Fragmente:
BIKES 1 : $\pi_{\text{BNr, BName, Bestand}}$(BIKES)

1. Variante:
VF_1: *BName*
VF_2: *Bestand, Preis, Typ*

	BName	Bestand	Preis	Typ
BName	45	45	0	0
Bestand	45	53	5	3
Preis	0	5	80	75
Typ	0	3	75	78

2. Variante:
VF_1: *BName, Bestand*
VF_2: *Preis, Typ*

Matrix CA

	BName	Bestand	Preis	Typ
BName	45	45	0	0
Bestand	45	53	5	3
Preis	0	5	80	75
Typ	0	3	75	78

3. Variante:
VF_1: *BName, Bestand, Preis*
VF_2: *Typ*

Matrix CA

	BName	Bestand	Preis	Typ
Name	45	45	0	0
Bestand	45	53	5	3
Preis	0	5	80	75
Typ	0	3	75	78

Abbildung 3.7: figure

BIKES 2 : $\pi_{\text{BNr, Preis, Typ}}$ (BIKES)

Verwenden der Matrix U und Acc

Matrix U

	BName	Preis	Typ	Bestand
q_1	1	0	0	1
q_2	0	1	0	1
q_3	0	1	1	0
q_4	0	0	1	1

Matrix Acc

	S_1	S_2	S_3
q_1	15	20	10
q_2	5	0	0
q_3	25	25	25
q_4	3	0	0

1. Variante:
$acc(VF_1) = 0$
$acc(VF_2) = 83$
$acc(VF_1, VF_2) = 45$
$sq = -45^2$

2. Variante:
$acc(VF_1) = 45$
$acc(VF_2) = 75$
$acc(VF_1, VF_2) = 8$
$sq = 45 * 75 - 8^2$

3. Variante:
$acc(VF_1) = 50$
$acc(VF_2) = 0$
$acc(VF_1, VF_2) = 78$
$sq = -78^2$

Variante 2 hat demnach die höchste Trennqualität.

Abbildung 3.8: figure

4 Verteilte Anfrage Verarbeitung

Beispiel Query: $KUN_1 = \pi_{KNR,KNAME}(KUN)$
$KUN_2 = \pi_{KNR,ORT}(KUN)$
$KUN = KUN_1 \bowtie KUN_2$

4.1 Query Processor

Umformen der SQl Queries in ausführbare Operationen für den Datenzugriff:

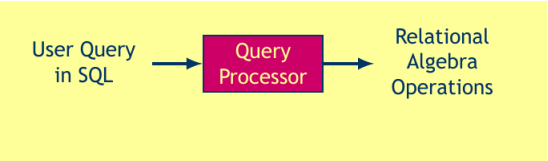

Abbildung 4.1: figure

4.2 Zentralisiert vs Dezentraliziert

Zentraliziertes System
- Umformen der SQL Anfrage in ausführbare Einheiten - Operationen der Rationalen Algebra.
- Auswählen des besten Ausführungsplans

verteiltes System
- berücksichtigen der Kommunikationskosten
- auswählen des besten Knotens.

Relationen werden mit den ersten drei Buchstaben von der DB Tabellenname zufkünfig benannt.

4.3 Überblick

4.3.1 Ziel der Optimierung

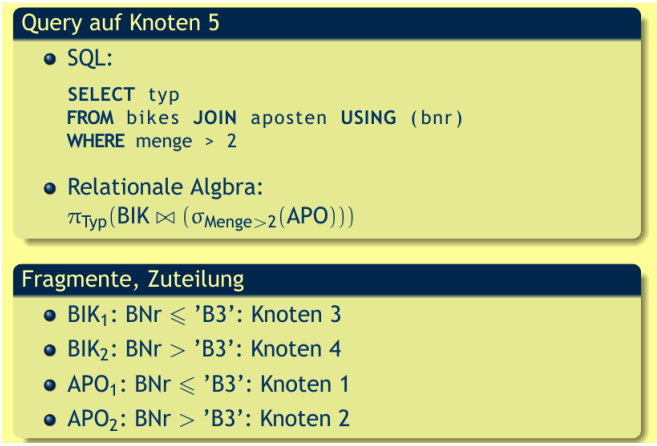

Abbildung 4.2: figure

- durch Umformen und Optimieren soll eine Kosenfunktion mimimiert werden
  - CPU Zeit
  - IO Zeit
  - Comm Zeit
- verschiedene Gewichte in unterschiedlich verteilten Umgebungen. WAN - Kommzeit LAN - ähnlich gewichtet.

4.3.2 Komplexität

$\theta\pi$ mit Duplikate : O(n)
π Ohne Duplikate - GROUP O(n log n)

$\bowtie \cup \cap \div$: O(n log n)
$\times$: $O(n^2)$

4.4 Methode

4.4.1 Verteilte Verarbeitung

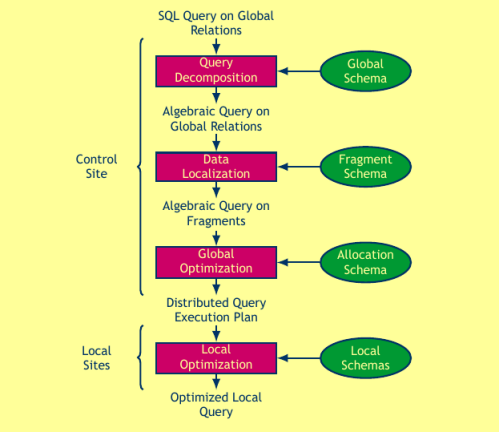

Abbildung 4.3: figure

4.4.2 Zerlegung

Sql Query zerlegen und umformen in relationale Algebra unter verwendung des globalen Schemas :

- Normalisierung (Bedingung in Where Klausel)
- Analyse um inkorrekte Queries zurückzuwechseln.

  Analyse nach Typ

  Analyse nach Semantik
- Vereinfachung, Redundanz beseitigen.
- Umformen in optimalen Ausdruck der relationalen Algrbra.

4.5 Lokalisierung

- verwenden des Fragmentierunggsschema.
- verteilte Anfrage mit globalen Relationen abbilden in Anfragen mit Fragmenten:

  Ersetzen der globalen Relation mit den Fragmenten :

  $\cup$ für Horizontale Fragmentierung

  $\bowtie$ für Vertikale Fragmentierung.
- Optimierng der lokalisierten Anfrage dürch Reduktion mit Selektion oder Join.

4.5.1 PHF

Beispiel

AUF(ANr, Datum, KNr) ist fragmentiert:

$AUF_1 = \sigma_{ANr \leqslant A3}(AUF)$
$AUF_2 = \sigma_{A3 < ANr \leqslant A6}(AUF)$
$AUF_3 = \sigma_{ANr > A6}(AUF)$

Rekonstruktion mit:

$AUF = AUF_1 \cup AUF_2 \cup AUF_3$

Abbildung 4.4: figure

Bemerkung: Eine Selektion auf einem Fragment mit einer Bedingung, die der Bedingung für die Fragmentierung wiederspricht , ergibt leere Resultatsrelationen.

4.5.2 Schritten für Lokalisierung PHF

Schritt 1 globaler Query Baum erstellen

Schritt 2 globale Relation mit Fragmenten ersetzen.

Schritt 3 Reduktion:
Eine Reduktion ist möglich wenn die Fragmentierung mit dem Join Attribute erfolgte. Wenn sich die Bedingungen der Fragmente wiedersprechen, resultiert eine leere Relation. Vor einer möglichen Reduktion wird das Distributivgesetz angewendet :
$(R_1 \cup R_2) \bowtie S = (R1 \bowtie S) \cup (R_2 \bowtie S)$

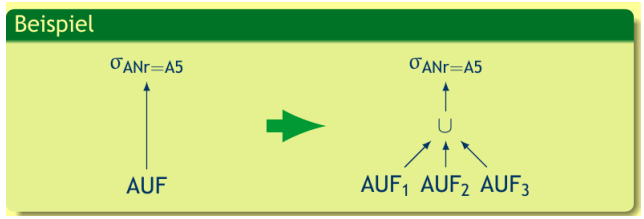

Abbildung 4.5: figure

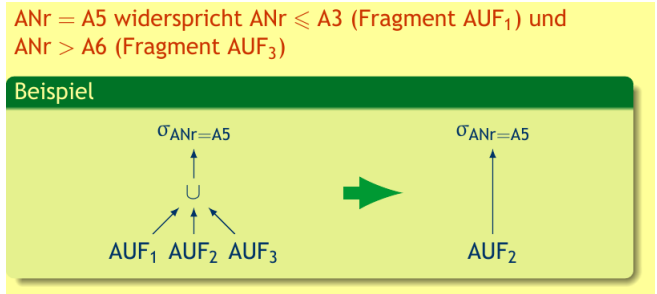

Abbildung 4.6: figure

4.5.3 Reduktion mit Join für PHF

4.6 Reduktion für VF

Eine Projektion auf einem Fragment ist nutzlos wenn die Attribute der Projektion nicht im Fragment enthalten sind.

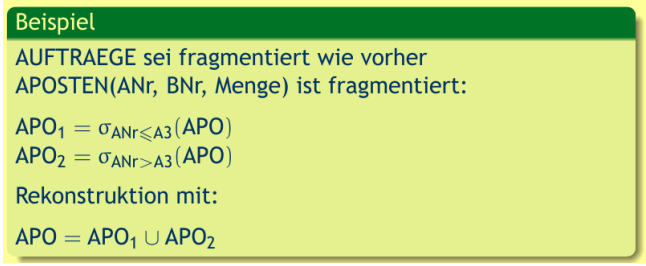

Abbildung 4.7: figure

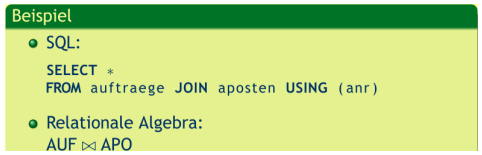

Abbildung 4.8: figure

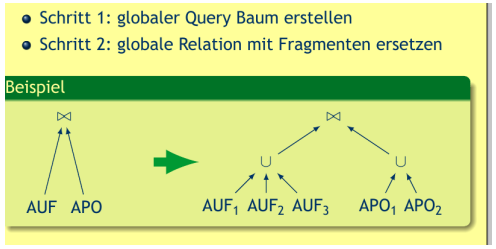

Abbildung 4.9: figure

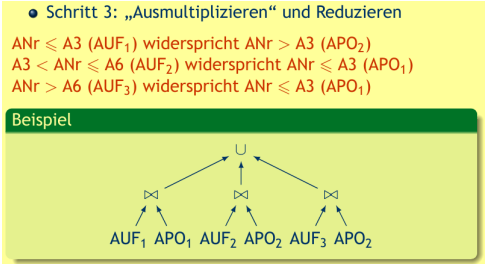

Abbildung 4.10: figure

Beispiel

KUNDEN(KNr, KName, Ort) ist fragmentiert:

$KUN_1 = \pi_{KNr,KName}(KUN)$
$KUN_2 = \pi_{KNr,Ort}(KUN)$

Rekonstruktion mit:

$KUN = KUN_1 \bowtie KUN_2$

Abbildung 4.11: figure

- Schritt 1: globaler Query Baum erstellen
- Schritt 2: globale Relation mit Fragmenten ersetzen
- Schritt 3: Reduktion

Beispiel

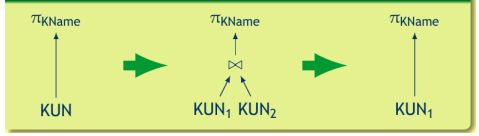

Abbildung 4.12: figure

Beispiel

KUNDEN(KNr, KName, Ort) ist fragmentiert:

$KUN_1 = \sigma_{Ort=Basel}(KUN)$
$KUN_2 = \sigma_{Ort \neq Basel}(KUN)$

AUFTRAEGE(ANr, Datum, KNr) ist abgeleitet fragmentiert:

$AUF_1 = AUF \triangleright KUN_1$
$AUF_2 = AUF \triangleright KUN_2$

Rekonstruktion mit:
$KUN = KUN_1 \cup KUN_2$
$AUF = AUF_1 \cup AUF_2$

Abbildung 4.13: figure

Anwenden der Join Reduktion

Beispiel

- SQL:

```
SELECT *
FROM kunden JOIN auftraege USING (knr)
WHERE ort = 'Olten'
```

- Relationale Algebra:
  $AUF \bowtie (\sigma_{Ort=Olten}(KUN))$

Abbildung 4.14: figure

5 Concurrency

5.1 Notation + Definitionen für Concurrency

Notation	Bedeutung
$r_i[x]$	Read(x) durch Transaktion T_i
$w_i[x]$	Write(x) durch Transaktion T_i
$o_i[x]$	Read(x) oder Write(x) durch T_i
c_i	Commit durch Transaktion T_i
a_i	Commit durch Transaktion T_i
e_i	Commit oder Abort durch T_i

Transaktion Folge von Read und Write Operationen auf beliebiegen Datenelementen x. Letzte Operation entweder Commit oder Abort.

Schedule,History ist eine foge von Datenbank Scheduler auf der Datenbank ausgeführten Lese und Screiboperationen.Es besteht aus Operationen verschiedener Transaktionen.Ordnung der Operationen innerhalb einer Transaktion muss beibehalten werden.

Serielle History Eine History H heisst seriell, wenn jeweils alle Operationen einer Transaktion direkt hintereinander ausgeführt werden (keine überlappte Ausführung der Transaktionen).

Konflikt Operation Zwei Datenbank-Operationen o_i und o_k heissen Konflikt-Operationen wenn :

- beide auf dasselbe Element x zugreifen.
- sie zu verschieden Transaktionen gehören.
- mindestens eine der beiden eine Schreiboperation ist.

Konflikt Serialisierbarkeit Eine History heisst konflikt-serialisierbar , wenn es eine serielle History mit gleicher Reihenfolge der Konflikt-Operationen gibt.

Recoverable History Eine History heisst Recoverable falls alle Commits der Transaktionen, von denen eine Transaktion T_i liest vor dem c_i erfolgen.

Vermeiden von cascading Aborts Eine History vermeidet cascading Aborts falls alle Commits der Transaktionen, von denen eine Transaktion T_i liest vor dem $r_i[x]$ erfolgen.

Strickte History Eine History heisst strikt falls jedes $r_i[x]$ oder $w_i[x]$ einer Transaktion T_i nach den Commits aller anderen Transaktionen erfolgt, die auf x geschreiben haben.

2 Phasen Sperrprotokoll Two Phase Lock : Eine Transaktion folgt dem 2 Phasen Sperrprotokoll

- vor einem Read bzw Write muss ein share Lock bzw exclusive lock verlangt und erlangt werden.
- die Sperre darf erst freigegeb werden wenn die zugehörige Operation vollständig durchgeführt worden ist.
- nach der ersten Freigabe einer Sperre darf keine neue Sperre verlangt werden.

Striktes 2 Phasen Sperrprotokoll • Alle Sperren müssen auf einmal mit dem Commit atomar freigegeben werden.

5.2 Einführung

Atomicity Alles oder Nichts

Consistency keine Verletzung von Integritätsregeln

Isolation keine Nebeneffekte bei Nebenläu gkeit

Durability Bestätigte Änderungen sind persistent

Transaktionen bieten :

- unteilbare und zuverlässige Ausführung auch bei Ausfällen
- korrekte Ausführung auch bei gleichzeitigem Zugriff durch mehrere Benutzer
- korrekte Verwaltung von Replikaten (falls unterstützt)

5.2.1 Aspekte der Transaktionsverarbeitung

- Transaktionsstruktur (Transaktionsmodell)

  flache Transaktion, verschachtelte Transaktion
- Konsistenzerhaltung der Datenbank
- Zuverlässigkeitsprotokolle

  Unteilbarkeit, Dauerhaftigkeit

  lokale Wiederherstellung

  globale Commit Protokolle
- Nebenläufigkeitskontrolle

  Ausführung nebenläufiger Transaktionen (Korrektheit?)

  Konsistenzerhaltung zwischen Transaktionen, Isolation
- Kontrolle der Replikate

  Kontrolle der gegenseitigen Konsistenz von Replikaten
- One-Copy Equivalence, ROWA

5.3 Arhitektur

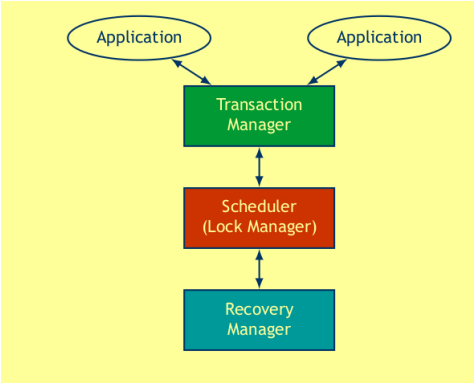

Abbildung 5.1: figure

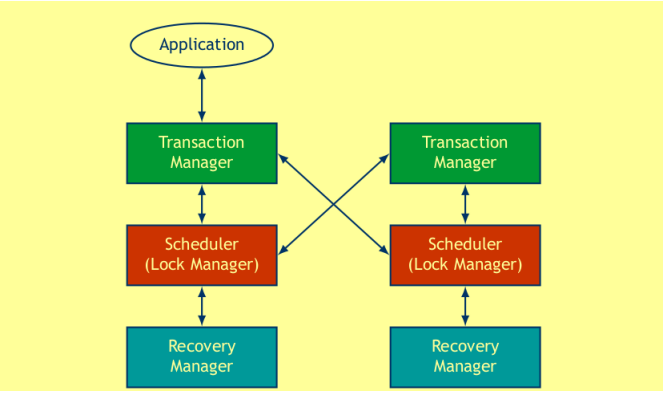

Abbildung 5.2: figure

5.4 Nebenläufigkeit

Nebenläufigkeitskontrolle wie in zentralisierten DBS

- Ablaufplan, History, Schedule
- serielle Ausführung, serielle History
- Äquivalenz von Histories

  Konflikt Äquivalenz

  Konflikt Operationen
- Serialisierbarkeit
- zwei verschiedene Arten Histories berücksichtigen

  lokale Ablaufpläne

  globaler Ablaufplan
- für Serialisierbarkeit des globalen Ablaufplans sind zwei Bedingungen nötig

  jeder lokale Ablaufplan muss serialisierbar sein

  zwei Konflikt Operationen müssen in der gleichen Reihenfolge auftreten in allen lokalen Ablaufplänen, in denen sie zusammen auftreten

```
Transaktion T1:          Transaktion T2:
Read(x)                  Read(x)
x := x - 100             Read(y)
Write(x)                 Commit
Read(y)
y := y + 100
Write(y)
Commit
```

- x liegt auf Knoten a, y liegt auf Knoten b
- LH_a, LH_b sind lokale serialisierbare (serielle) Ablaufpläne
  - $LH_a = R_1[x]\ W_1[x]\ R_2[x]$
  - $LH_b = R_2[y]\ R_1[y]\ W_1[y]$
- es gibt dazu aber keinen globalen seriellen Ablaufplan

Abbildung 5.3: figure

5.4.1 Beispiel : Global nicht serialisierbar

serialisierbar auf Knoten aber global nicht wegen verscheidene Reihenfolge.Wir wissen im Zentral wie man es macht, aber

5.5 Realisierung

- Basierend auf 2 Phasen Sperrprotokoll
  - Primary Copy 2PL.
  - Verteiltes 2PL
- häufig mit Snapshot verfahren.
  - Read Constency bei Oracle
  - verwenden der SCN Synchronization bei verteilten DBS.

SCN (System Change Number) is a primary mechanism to maintain data consistency in Oracle database. SCN is used primarily in the following areas, of course, this is not a complete list:

1. Every redo record has an SCN version of the redo record in the redo header (and redo records can have non-unique SCN). Given redo records from two threads (as in the case of RAC), Recovery will order them in SCN order, essentially maintaining a strict sequential order. As explained in my **paper**, every redo record has multiple change vectors too.
2. Every data block also has block SCN (aka block version). In addition to that, a change vector in a redo record also has expected block SCN. This means that a change vector can be applied to one and only version of the block. Code checks if the target SCN in a change vector is matching with the block SCN before applying the redo record. If there is a mismatch, corruption errors are thrown.
3. Read consistency also uses SCN. Every query has query environment which includes an SCN at the start of the query. A session can see the transactional changes only if that transaction commit SCN is lower then the query environment SCN.
4. Commit. Every commit will generate SCN, aka commit SCN, that marks a transaction boundary. Group commits are possible too.

Abbildung 5.4: figure

5.5.1 Zentrales 2PL

- einzelner Knoten verwaltet alle Sperrinformation.
- ein Lock manager fur das gesmate DDBMS
- jede Sperre muss beim zentralen Lock manger verlangt werden.
- einfach zu relisieren.
- Flachenhäls, weniger zuverlässig.

5.5.2 Primary Copy 2pl

- Lock Manager auf einigen Knoten verteilt
- jeder Lock Manager verantowrtlich für die Sperren einer Menge von Daten
- bei Replikaten wird eine Kopie als Primary Copy bestimmt, die anderen sind dann slave copies.
- Schreibsperre nur auf Primary Copy.
- nach Änderung der Primary Copy,Änderungen zu den slaves bringen.
- komplexeres Deadlock Handling
- weniger Kimmunikation bessere Performance.

5.5.3 Verteiltes 2PL

- Jeder Lock manager auf allen Knoten verteilt.
- jeder Lock anager vrantwortlich für die Sperren siner Daten
- ohne Replikate gleich wie Primary Copy 2PL
- mit Replikaten wird Read One Write All Protokoll implementiert.
  - irgendiene Kope kann zm Lesen verwendet werden
  - alle Kopien erhalten Schreibsperre vor Änderung.
- *komplexeres Daeadlock Handling*
- *höhere Kommunikationskosten als Primary Copy 2PL*

5.6 Deadlocks

Transaktionen sind im einem Deadlock wenn sie `blockiert` sind und **es bleiben bis das System eingreift.**
Deadlocks treten in 2PL auf zur Vermeidung von nicht serialiserbare Aufbaupläne. Um einen Deadlock aufzulösen muss eine Transaktion abgebrochen werden. In einem **Wait for Graph oder WFG** können Deadlocks erkannt werden.

WFG ist ein gerichtete Graph und dtellt die warte auf Beziehung zwischen Transaktionen dar. Transaktionen sind die Knoten im WFG.

> Kante $T_i \leftarrow T_j$ beduetet T_i wartet auf die Freigabe der Sperre auf T_j Zyklen in WFG sind dann deadlocks.

5.6.1 Deadlock Erkennung

- Ansatz automatisch Deadlocks erkennen und eine Betroffene Transaktion abbrechen.
- Bevorzugter einsatz braucht viel Resourcen ist aber ein einfaches Verfahren.
- Timeout: Transaktion die zu lange blockiert ist abbrechen.
  - einfach zu realiseren.
  - bricht Transaktionen unnötigerweise an.
  - Deadlocks können lange bestehen.

WFG kann benutzt werden um :

- Deadlock zu finden -
  - wird eine Transaktion blockiert, Kante in WFG einfügen
  - WFG periodisch für Zyklen durchsuchen.
- Wird ein Deadlock ekannt, eine Transaktion im Zyklus wählen und abbrechen.

5.6.2 Verteilte Deadlocks

- Wenn Deadlock zwi Knoten umfasst, kann ihn **keiner der Knoten erkennen.**
- Problem lokale und globaler WFG.
- Szenario : t_1 und T_2 auf Knoten 1 und T_3, T_4 auf Konten 2. $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_1$

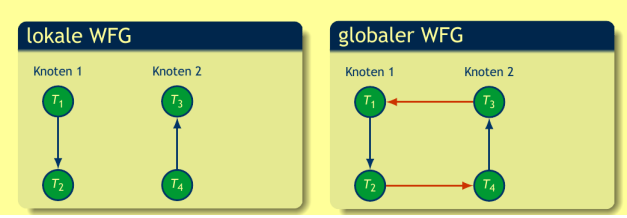

Abbildung 5.5: figure

6 Verteilte Transaktionen 2

Eine Datenbank ist zuverlässig wenn sie robust und zur Widerherstellung fähig ist. Diese Idee steckt auch in den ACID eigenschaften :

Atomicity requires that each transaction is äll or nothing": if one part of the transaction fails, the entire transaction fails, and the database state is left unchanged. An atomic system must guarantee atomicity in each and every situation, including power failures, errors, and crashes. To the outside world, a committed transaction appears (by its effects on the database) to be indivisible (ätomic"), and an aborted transaction does not happen.

Consistency The consistency property ensures that any transaction will bring the database from one valid state to another. Any data written to the database must be valid according to all defined rules, including but not limited to constraints, cascades, triggers, and any combination thereof. This does not guarantee correctness of the transaction in all ways the application programmer might have wanted (that is the responsibility of application-level code) but merely that any programming errors do not violate any defined rules.

Isolation The isolation property ensures that the concurrent execution of transactions results in a system state that would be obtained if transactions were executed serially, i.e. one after the other. Providing isolation is the main goal of concurrency control. Depending on concurrency control method, the effects of an incomplete transaction might not even be visible to another transaction.[citation needed]

Durability Durability means that once a transaction has been committed, it will remain so, even in the event of power loss, crashes, or errors. In a relational database, for instance, once a group of SQL statements execute, the results need to be stored permanently (even if the database crashes immediately thereafter). To defend against power loss, transactions (or their effects) must be recorded in a non-volatile memory.

Wir werden auf `Atomicity` und `Durability` fokusieren.
Fehlerarten:

Transaktionsfehler Transaktionssabbruch, einseitig, Deadlock

System oder Kontenausfall Ausfall von CPU , Haputsapeicher , Stromversogung - Hauptspeicherintervale gehen verloren. Teil oder Totalausfall.

Plattenfehler Plattenfehler führen zu verlust gespeicherte Daten. Headcrash, Ausfall des Controllers.

Verbindungsausfall Verlotene oder nicht zuteilbare Nachrichten. Trennung des Netzwerks.

6.1 Lokale oder Verteilte Zuverlässigkeit

Zentrales DBS Behandelt transaktionsfehler, System oder Knotenausfall und Plattenfehler. Bei Systemausfällen oder Plattenfehler handelt es immer um einen vollständigen Ausfall.

Verteiltes DBS Behandelt zusätzlich koordination bei Transaktionsfehler, System oder Knotenausfall, Plattenfehler *Verbindungsausfall, erhöhte Verfügbarkeit.*

6.2 Lokale Widerherstellung

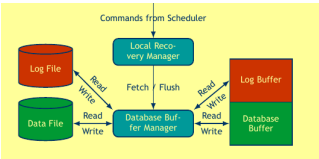

Abbildung 6.1: figure

Einzelnbe DBS sind fähig zur Widerherstellung durch :

- angemessene Schreibstrategie - force oder no force.
- Angemessene Seitenersetzung (Paging Stategy) - Steal oder No Steal.

- Protokollierung von Änderungen - undo/redo.
- Write Ahead Logging (WAL)

  Vor Commit (Redo) Log Einträge sichern.

  Vor Änderung der permanenten Datenbank (Undo) Log einträge sichern.
- Robustes verfahren benutzen

  Undo, Redo indempotent.

  ARIES

6.3 Zuverlässigkeit

- Sicherstellen von Atomicity und Durability verteilter Transaktionen.
- Jeder Knoten kann

  lokaler Teil der Transaktion zuverlässig verarbeiten

  nach Bedarf lokales Commit, Rollback, Recovery durchführen.
- Übergeordnete Protokolle müssen sich mit der Koordination der beiteiligten Knoten befassen.

  Start der Transaktion beim Ursprungsknoten verarbeiten

  Read und Write an den Zielknoten verarbeiten, spezielle vorkehrungen bei Replikation.

  Abort, Commit, und Recover spezifisch für DDB.

6.3.1 Komponenten

Unterschieden in :

Coordinator Prozess beim Ursprungsknoten, steuert die Ausführung.

Participant Prozess bei den anderen Knoten die an der Transaktion beteiligt sind.

Protokolle:

Commit Protokolle Wie das Commit einer Transaktion verarbeiten wenn mehr als ein Knoten am Commit beteiligt ist? Regelt Atomicity und Durability einer verteilten Transaktion

Termination Protokolle Verwendet von Knoten, die vom Ausfall anderer Knoten betroffen sind. Regelt für einen Knoten das Commit/Abort, wenn andere Knoten ausfallen. Blocking/Non-Blocking: Müssen Knoten auf Recovery anderer Knoten warten, um Transaktion abzuschliessen?

Recovery Protokolle Verwendet von Knoten, die ausgefallen sind. Regelt das Recoveryverfahren beim Wiederanlauf nach einem Ausfall Unabhängig: Ein ausgefallener Knoten kann das Ergebnis einer Transaktion bestimmen, ohne Information von anderen Knoten zu haben.

6.4 2 Phase Commit

1. Choice: Der Koordinator fragt alle Teilnehmer, ob sie bereit sind für ein Commit: Prepare. Jeder Teilnehmer teilt dem Koordinator seine Entscheidung mit: Vote-Commit oder Vote-Abort.
2. Descision:Der Koordinator trifft endgültige Entscheidung: Commit: Falls alle Teilnehmer bereit zum Commit Abort: In allen anderen Fällen und teilt sie den Teilnehmern mit: Global-Commit oder Global-Abort. Diese müssen sich entsprechend verhalten und bestätigen: Acknowledge.

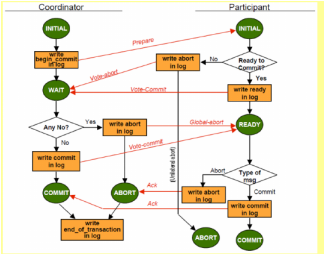

Abbildung 6.2: figure

Beobachtungen

1. Ein Teilnehmer kann einseitig Abort durchführen, vorausgesetzt, er hat nicht mit Vote-Commit geantwortet.
2. Nachdem ein Teilnehmer mit Vote-Commit geantwortet hat, muss er zum Commit bereit sein und kann nicht mehr umentscheiden.
3. Ein Teilnehmer im Ready-Zustand muss bereit sein zum Commit oder Abort, je nach Entscheid des Koordinators.

4. Die endgültige Entscheidung ist Commit, falls alle Teilnehmer Vote-Commit geantwortet haben, oder Abort, falls irgendein Teilnehmer mit Vote-Abort geantwortet hat.
5. Koordinator und Teilnehmer verwenden Time-Out Methoden, um möglichen Wartezustand zu verlassen.

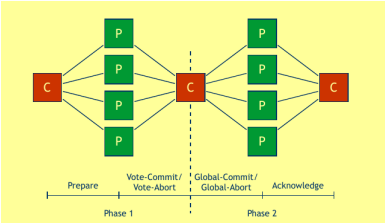

Abbildung 6.3: Zentraliziertes 2pc

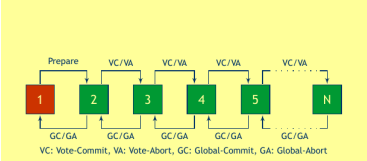

Abbildung 6.4: Lineares 2PC

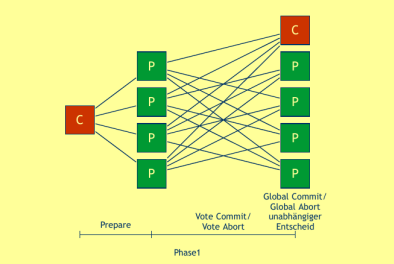

Abbildung 6.5: Verteiltes 2PC

6.4.1 Zustandsübergänge

- Time out des Koordinates im Zustand `wait`,`abort`,`commit` möglich.

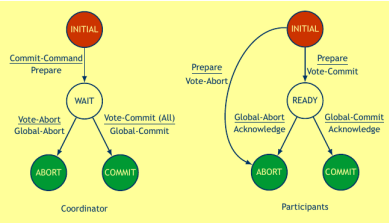

Abbildung 6.6: Zustandsübergänge 2PC

- `wait` Zustand :

  Koordinator wartet auf Entscheid der Teilnehmer

  Lösung: Koordinator entscheidet Global-Abort der Transaktion.

- `commit` oder `abort` :

  Koordinator ist nicht sicher ob das Cimmit oder Abort verfahren durch die LRMs[1] alle Teilnehmer durchgeführt wurde.

  Lösung: Global commit oder Global Abort an alle Teilnehmer die nicht geantwortet haben, erneut senden.

- Time-Out eines Teilnehmers im Zustand `initial` oder `ready` möglich.
- `initial` Zustand :

  Teilnehmer wartet auf das Prepare des Koordinators.

  Lösung: Teilnehmer führt einseitig Abort durch. Falls Prepare später ankommt, mit Vote Abort antworten oder ignorieren.

- `ready` Zustand :

  Teilnehmer darf nach dem Vote Commit sich nicht umentscheiden und einseitig Abort durchführen.

  Lösung: Teilnehmer ist blockiert und muss auf den entgültigen Entscheid des Koordinators warten.

6.4.2 Recovery Protocols

Protokolle die ein Koordinator oder Teilnehmer nach einem Ausfall verwenden für die Wiederherstellung beim Neustart.
Annahmen:

[1] Local Recovery Manager

1. Logbeschreibungen und Versenden von Nachrichten sind atomare Aktionen.
2. Zustandsübergänge erfolgen nach Versenden der Nachricht.

6.4.3 Ausfälle

Koordinator

- Koordinator im Zustand `initial` fällt aus - neustart der Transaktion
- `wait` - Neustart des Commit Verfahrens mit nochmailigem Versenden der Prepare Nachricht.
- `commit` oder `abort`

  Falls alle Acknowledge Meldungen erhalten sind nichts tun sonnst Termination Protokolls answenden.

Teilnehmer

- `initial` Nach der Wiederherstellung führt Teilnehmer einsieitig ein Abort der Transaktion durch.
- `ready` Ablauf wie nach Time out im `ready` Zustand und Termination Protocols anwenden.
- `commit, abort` Nichts tun.

6.4.4 Probleme mit 2pc

- Blockierend. Im Zustand `ready` muss Teilnehmer auf Entscheid des Koordinators warten. Fällt Koordinator aus, ist Teilnehmer blockiert. Verfügbarkeit reduziert.
- Recovery eines Teilnehmers nach Ausfall im Zustand `ready` kann nicht unabhängig erfolgen.Verfügbarkeit des Koordinators oder eines Teilnehmers, der die Rolle des Koordinators übernimmt ist nötig.

6.5 3 Phase Commit

- Nicht Blockierend.
- Commit Protocol ist **nicht blockierend genau dann wenn:**

  Synchron innerhalb Zustandübergang

  Keine Zustände hat die benachbahrt sind zu einem Commit und Abort Zustand. Oder die die zu einem Commit zustand *nicht* benachbart sind aber nicht commitable sind.

  Committable : Zustand in dem sichergestellt ist, dass alle Teilnehmer mit Vote-Commit geantwortet haben

7 Replikation I

7.1 Einleitung

7.1.1 Grunde für Replikation

Züverlassigkeit vemeidet ßingle points of failiure."

Perormance Vermeidet Kommunikationskosten durch lokalen Zugriffe.

Skalierbarkeit Unterstützt Wachstum des Systems

Andforderungen durch Anwendungen Spezifikation??

Probleme
- Transparenz der Replikation - Abbilden logische Zugriffe in Physische Zugriffe auf Kopieen.
- Fragen der Konsistenz : Kritieren und Sync der Kopieen.

7.1.2 Ausführungsmodell

Es gibt physische Kopien der logischen Objekte im System. Zugriffe betreffen logische Objekte, werden aber in Zugriffe auf pysische Objekte umgesetzt.:

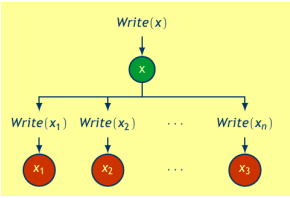

Abbildung 7.1: figure

7.2 Konsistenzmodelle

Wie wird die Konsiszenz des globalen DB Zustands definiert?

- Transaktionsbezogene Kritieren :

  One Copy Serializability

  Snapshot Isolation

- Datenbezeogene Kritieren : Eventual Consistency
- Begrentzte Inkonsistenz : Epsilon Serializability

Mit welchen Verfahren wird geneseitige Konsistenz der Replikate erreicht?

Transaktionsbezoegene Kritieren (Strong Consistency)

- One Copy Equivalenz : Gegenseitige Konsistenz wenn alle Kopien identische Werte haben.Wirkung einer Transaktion auf replizierte Daten ist die gleiche, wenn sie auf einder einzelnen Datenmenge operierte.
- One Copy Serializability : Wirkung von Transaktionen auf replizierte Daten ist die gleiche wenn sie eine nach der anderen auf eine einzelnen Datenmenge operierten. Histories sind äquivalenz zu einer seriellen Ausführung auf nicht replizierten Daten.

Datenbezogene Kritieren

- Angeschwächte formen der Konsistenz:
- Eventual Consistency : letzendlich oder später erreicht. Wenn weitere Updates ausblieben kovergieren die Replikate zu identischen Kopien.
- Nur die Ausbreitung der Updates muss guarantiert sein. Das ist kein problem solange die Nutzer immer auf das gleiche Replikat zugreifen :

  Übergang zu Client-Centric Consistency. Guarantiert einem einzelnen Nutzer konsistenten Zugriff auf daten.

Client Centric Consistency

- Monotonic Reads - (gleichbleibend) Liest ein Prozess Datenelement x, dann gibt jedes nachfolgende Lesen von x durch diesen Prozess denselben oder neueren Wert zurück.
- Monotonic Writes - Schreiboperation auf Datenelement x ist abgeschlossen, bevor derselbe Prozess nachfolgende Schreiboperationen auf x ausführt.
- Read your Writes - Die Wirkung von Schreiboperation auf Datenelement x wird immer in nachfolgenden Leseoperationen auf x durch denselben Prozesse gesehen.
- Writes follow Reads - Schreiboperationen auf Datenelement x, die auf ein Lesen durch denselben Prozess folgen, überschreiben immer denselben oder neueren Wert von x.

7.3 Update Propogation Strategies

Können entweder `Eager(sync)` oder `Lazy(async)` im kombination mit `Primary Copy (Master)` oder `Update Everywhere (Group)`.

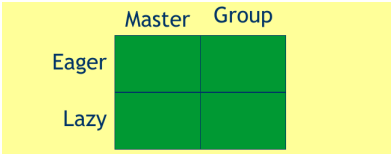

Abbildung 7.2: figure

Eager(Sync Replikation) Jede Änderung wird sofot zu allen Kopien übertragen. Übertragung der Änderungen erfolgt innerhalb der Grenzen der Transaktion.ACID Eignenschaften gelten für alle Kopien.

Lazy(Async Replikation) Zuerst werden die lokalen Kopein geändert. Anschliessend werden die Änderungen zu allen anderen Kopien übertragen *Push/Pull*. Während der Übertraging sind Kopien inkonsistent. Zeitraum der Inkonsistenz kann in Abhängigkeit der Anwendung angepasst werden.

Master/Primary Copy Replication Es gibt eine einzigge Kopie auf der Änderungen ausgeführt werden können (Master). Alle anderen Kopien (Slaves) übernehmen die Änderungen vom Master. Für verschiedene Datenelemente können verschidene knoten Master sein.

Group/Update everywhere Replication Änderungen können auf jeder Kopie ausgeführt werden. D.h jeder Knoten der eine Kopie besitzt, kann auf ihr Änderungen ausführen.

7.4 Replikationsstrategieen

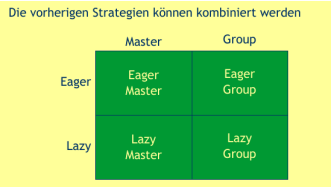

Abbildung 7.3: figure

Eager Master (Synchronous Primary Copy) Replication

- Primary Copy

  Read Lokales lesen (eigene Kopie), Resultat zurückgeben.

  Write lokales Schreiben, **Write an alle slaves weiterleiten (FIFO/Timestamp reihenfolge)**, Kontrolle sofort dem Nutzer zurückgeben.

  Commit verwende 2PC als Koordinator

  Abort lokales Abort, slaves informieren.
- Slave:

  Read Lokales lesen (eigene Kopie), Resultat zurückgeben.

  Write Write von Master : ausführen der writes in richtiger Riehenfolge,**Writes von Client:** zurückweisen oder an Master wieterleiten.

  Slave ist Teilnehmer an 2PC der Transaktionen vom Master.
- Vorteile :

  Änderungen müssen nicht koordiniert werden.

  Keine Inkonsistenzen.
- Nachteile:

  Längste Antwortzeit

Nur bei wenigen Updates sinnvoll (Master ist Flaschenhals)

Lokale Kopien sind beinahe nutzlos

selten eingesetzt

Eager Group (Synchronous Update Everywhere) Replication

- Read One Write All (ROWA)

  jeder Knoten verwendet 2 Phasen Sperrprotokoll

  Leseoperationen werden lokal durchgeführt

  Schreiboperationen werden auf allen Knoten ausgeführt mithilfe eines verteilten Sperrprotokolls
- Vorteile:

  Keine Inkonsistenzen

  elegante Lösung (symmetrisch)
- Nachteile:

  Vielzahl von Nachrichten

  Antwortzeiten der Transaktionen sind sehr lang

  beschränkte Skalierbarkeit (Deadlock Wahrscheinlichkeit wächst mit Anzahl Knoten)

Lazy Master (Asynchronous Primary Copy) Replication

- Primary Copy

  Read: lokales Lesen (eigene Kopie), Resultat zurückgeben

  Write: lokales Schreiben Kontrolle dem Nutzer zurückgeben

  Commit, Abort: Transaktion lokal beenden Irgendwann nach dem Commit: an alle Knoten die Änderungen in einer einzigen Nachricht übermitteln (FIFO oder Timestamp Reihenfolge)
- Slave

  Read: lokales Lesen (eigene Kopie), Resultat zurückgeben

  Nachricht von Master: Änderungen in richtiger Reihenfolge (FIFO oder Timestamp) anwenden

  Write von Client: zurückweisen oder an Master weiterleiten

  Commit, Abort: nur für lokale Read-only Transaktionen
- Vorteile:

  Keine Koordination nötig

  kurze Antwortzeiten (Transaktionen sind lokal)
- Nachteile

  lokale Kopien sind nicht aktuell

  Inkonsistenzen (verschiedene Knoten haben unterschiedliche Werte für das gleiche Datenelement)

Lazy Group (Asynchronous Update Everywhere) Replication jeder Knoten:

- textbfRead: lokales Lesen (eigene Kopie), Resultat zurückgeben
- **Write:** lokales Schreiben Kontrolle dem Nutzer zurückgeben
- **Commit, Abort:** Transaktion lokal beenden Irgendwann nach dem Commit: an alle Knoten die Änderungen in einer einzigen Nachricht übermitteln (FIFO oder Timestamp Reihenfolge)
- Nachricht von anderem Knoten:

  Erkennen von Konflikten

  Änderungen anwenden

  Reconciliation (Abgleichen)
- Vorteile:

  Keine Koordination nötig

  kürzeste Antwortzeiten
- Nachteile

  Inkonsistenzen

  Änderungen könne verloren gehen (Abgleich)

8 Nosql

8.1 Einführung

A blogger, often referred to as having made the term popular is Rackspace employee Eric Evans who later described the ambition of the NoSQL movement as "the whole point of seeking alternatives is that you need to solve a problem that relational databases are a bad fit for" (cf. [Eva09b]). This section will discuss rationales of practitioners for developing and using nonrelational databases and display theoretical work in this field. Furthermore, it will treat the origins and main drivers of the NoSQL movement.

8.1.1 Motivation

Avoidance of unneeded complexity: databases provide a variety of features and strict data consistency. But this rich feature set and the ACID properties implemented by RDBMSs might be more than necessary for particular applications and use cases. As an example, Adobe's ConnectNow holds three copies of user session data; these replicas do not neither have to undergo all consistency checks of a relational database management systems nor do they have to be persisted. Hence, it is fully sufficient to hold them in memory (cf. [Com09b]).

High Throughput Some NoSQL databases provide a significantly higher data throughput than traditional RDBMSs. For instance, the column-store Hypertable which pursues Google's Bigtable approach allows the local search engine Zvent to store one billion data cells per day [Jud09]. To give another example, Google is able to process 20 petabyte a day stored in Bigtable via it's MapReduce approach [Com09b].

Horizontal Scalability and Running on Commodity Hardware NoSql trying to address problems such as Data Scaling, Performance of single servers, Rigid schema design.Nosql databases are designed to scale well in the horizontal direction and not rely on highly available hardware. Sharding is easier.

Avoidance of expensive Object Relational Mapping Most of the NoSQL databases are designed to store data structures that are either simple or more similar to the ones of object-oriented program- ming languages compared to relational data structures. They do not make expensive object-relational mapping necessary (such as Key/Value-Stores or Document-Stores). This is particularly important for applications with data structures of low complexity that can hardly benefit from the features of a re- lational database. Dare Obasanjo claims a little provokingly that "all you really need [as a web de- veloper] is a key<->value or tuple store that supports some level of query functionality and has de- cent persistence semantics." (cf. [Oba09a]). The blogger and database-analyst Curt Monash iterates on this aspect: "SQL is an awkward fit for procedural code, and almost all code is procedural. [For data upon which users expect to do heavy, repeated manipulations, the cost of mapping data into SQL is] well worth paying [. . .] But when your database structure is very, very simple, SQL may not seem that beneficial." Jon Travis, an engineer at SpringSource agrees with that: "Relational databases give you too much. They force you to twist your object data to fit a RDBMS." (cited in [Com09a]).

Complexity and Cost of Setting up Database Clusters See easier sharding.

Compromising Reliability for better performance "different scenarios where applications would be willing to compromise reliability for better performance." As an example of such a scenario favoring performance over reliability, he mentions HTTP session data which "needs to be shared between various web servers but since the data is transient in nature (it goes away when the user logs off) there is no need to store it in persistent storage."

Porblems with one size fits all thinking

1. Continuous growth of data
2. Process large amounts of data in short amounts of time.
3. Social media example - unrelated islands of data, low user/transaction value - no need for string data integrity.

Myth of Efffortless Dist and Partitioning of centralised data models and nor work correct any longer. The professionals of Ajatus agree with this in a blog post stating that if a database grows, at first, replication is configured. In addition, as the amount of data grows further, the database is sharded by expensive system admins requiring large financial sums or a fortune worth of money for commercial DBMS-vendors are needed to operate the sharded database (cf. [Aja09]).Shalom concludes that in his opinion relational database management systems will not

disappear soon. However, there is definitely a place for more specialized solutions as a "one size fits all" thinking was and is wrong with regards to databases.

Developments in Programming ORMs[1] hide SQL in many frameworks.The NoSQL databases react on this trend and try to provide data structures in their APIs that are closer to the ones of programming languages (e. g. key/value-structures, documents, graphs).

Cloud Computing

1. High -> inf scalability.
2. Low admin/overhead.

Following databases work well:

- Data warehousing specific dbs
- Simple scalable and fast key value stores.
-

The RDBMS plus Caching-Layer Pattern/Workaround vs. Systems Built from Scratch with Scalability in Mind : As scalability requirements grow and these technologies are less and less capable to suit with them. In addition, as NoSQL datastores are arising Hoff comes to the conclusion that "[with] a little perspective, it's clear the MySQL + memcached era is passing. It will stick around for a while. Old technologies seldom fade away completely." (cf. [Hof10c]). As examples, he cites big websites and players that have moved towards non-relational datastores including LinkedIn, Amazon, Digg and Twitter. Hoff mentions the following reasons for using NoSQL solutions which have been explained earlier in this paper:

- Relational databases place computation on reads, which is considered wrong for large-scale web applications such as Digg. NoSQL databases therefore do not offer or avoid complex read operations.
- The serial nature of applications1 often waiting for I/O from the data store which does no good to scalability and low response times.
- Huge amounts of data and a high growth factor lead Twitter towards facilitating Cassandra, which is designed to operate with large scale data.
- Furthermore, operational costs of running and maintaining systems like Twitter escalate. Web ap- plications of this size therefore "need a system that can grow in a more automated fashion and be highly available." (cited in [Hof10c]).

[1] Object Relational Mappers

Yesterday vs Today by Stonebraker, see below). In the 1960s and 1970s databases have been designed for single, large high- end machines. In contrast to this, today, many large (web) companies use commodity hardware which will predictably fail. Applications are consequently designed to handle such failures which are considered the "standard mode of operation", as Amazon refers to it (cf. [DHJ+ 07, p. 205]). Furthermore, relational databases fit well for data that is rigidly structured with relations and allows for dynamic queries expressed in a sophisticated language. Lehnhardt and Lang point out that today, particularly in the web sector, data is neither rigidly structured nor are dynamic quieries needed as most applications already use prepared statements or stored procedures. Therefore, it is sufficient to predefine queries within the database and assign values to their variables dynamically (cf. [PLL09]). Furthermore, relational databases were initially designed for centralized deployments and not for distribu- tion. Although enhancements for clustering have been added on top of them it still leaks through that traditional were not designed having distribution concepts in mind at the beginning (like the issues adverted by the "fallaclies of network computing" quoted below). As an example, synchronization is often not im- plemented efficiently but requires expensive protocols like two or three phase commit. Another difficulty Lehnhardt and Lang see is that clusters of relational databases try to be "transparent" towards applications. This means that the application should not contain any notion if talking to a singly machine or a cluster since all distribution aspects are tried to be hidden from the application. They question this approach to keep the application unaware of all consequences of distribution that are e. g. stated in the famous eight fallacies of distributed computing (cf. [Gos07]2 :

1. The network is reliable
2. Latency is zero
3. Bandwidth is infinite
4. The network is secure
5. Topology doesn't change
6. There is one administrator
7. Transport cost is zero
8. The network is homogeneous

8.2 Cassandra

1. **Welches sind die wichtigsten Gründe, die zur Entwicklung Ihres NoSQL Systems führten? Welche Anforderungen erfüllen diese Systeme?**

- Verarbeitung von grossen Datenmengen
- Availability
- Reliability
- Kein Single Point of Failure
- Läuft auf Cheap Comodity Hardware
- High Write Output
- Inbox Search Support (Suche durch die Facebook Inbox)
- Geographisch verteilt

2. **Welches „logische“ Datenmodell bietet das System? Welches API bietet das System? Wie werden Daten manipuliert und abgefragt?**

Datenmodell: Eine Instanz von Cassadra besteht aus nur 1 Tabelle, die eine verteilte, multidimensionale Map bezeichnet, die durch einen Schlüssel indexiert ist. Schlüssel: 16-36 Bytes, Value: Object. Struktur:
Zeilen mit String als Schlüssel
Spalten: haben Namen und speichern eine Anzahl Werte, die durch einen Zeitstempel identifiziert sind. Jede Zeile kann eine unterschiedliche Zahl von Spalten haben.
Spaltenfamilien: willkürliche Zahl Spalten pro Zeile können zu Familien zusammengefasst werden. Es können beliebig viele Familien spezifiziert werden, aber effektiv nur wenige. Familien können dynamisch zur Laufzeit mit Spalten und Superspalten ergänzt werden.
-Superspalten: weitere Gruppierung: Name und willkürliche Anzahl Spalten Ansprache von Daten über Tripel (Zeilenschlüssel, Spaltenschlüssel, Zeitstempel). Zeilenschlüssel über column-family:supercolumn:column Sortierung nach Zeitstempel oder Name möglich

API: drei Operationen:
- get(table, key, columnName)
- insert(table, key, rowMutation)
- delete(table, key, columnName) à Familie, Superzeile, Zeile
Operationen sind atomar per replica, Anzahl betroffene Spalten ist egal.
-read: Konsistenzgarantie kann spezifiziert werden (nächster Knoten, Antworten verschiedener Knoten)
-insert/update: „a quorum of replica nodes has to acknowledge the completion of the writes“. Weiterleitung zu willkürlichem Server eines Cassandra Clusters, so dass Partitionierung und Redistribution ermöglicht werden.

3. **Verteilungsarchitektur**
Ringarchitektur mit allen Knoten. Jedem Knoten ist ein bestimmter Range der Hashkeys zugeteilt. Replikation
SimpleStrategy: Replikation auf den N-1 nächsten Knoten
NetworkTopologyArchiteture: definierbare Anzahl Replikate pro Datencenter

4. **Wie werden Daten repliziert? Wie wird die Konsistenz der Daten sichergestellt?**

- Auf jeder Instanz ist ein Replikationsfaktor N definiert, wobei alle Daten auf N Hosts repliziert werden
- Jeder Schlüsser wird an einen Coordinator Nodeßugewiesen, welcher einerseits die Daten dazu lokal speichert und andererseits die Daten auf N-1 Knoten verteilt
- Daten können nach verschiedenen Richtlinien verteilt werden z.B. Rack Unaware, Rack Aware und Datacenter Aware um Risiken besser zu vermindern
- Cassandra bestimmt mit hilfe von Zookeeper einen Leader, welcher die Verantwortlichkeit für Datenbereiche an die Knoten verteilt
- Metadaten werden lokal und fehlertolerant in Zookeeper gespeichert
- Die Daten werden über mehrere Datacenter repliziert und sollten auch bei Ausfall eines ganzen Datacenters verfügbar sein.
- Konsistenz wird sichergestellt indem ein Quorum festgelegt wird, wieviel Knoten einen Schreibvorgang bestätigen müssen, auch für Lesevorgänge kann ein Quorum festgelegt werden, muss aber nicht

5. **Welche Vorkehrungen werden für die Verfügbarkeit und Ausfallsicherheit getroffen?**

- Daten bzw. Nodes sind geographisch verteilt und werden über mehrere Datencenter repliziert. Dadurch wird gewährleistet, dass die Latenzzeiten klein gehalten werden und der Ausfall eines Knotenpunktes keinen Einfluss auf das System hat. Durch das abschwächen des Quorum-System, der die Konsistenz aufrecht erhalten soll, wird die Beständigkeit des Systems garantiert.
- Fehlerdetektion durch Accrual Failure Detector: Jedem Knoten wird ein Wert zugeteilt, anhand dessen man abschätzen kann, wie hoch die Chance ist, dass dieser Knoten ausfällt, statt diesem Knoten einen boolean-Wert zuzuteilen.

- Der Knoten wählt beim Starten ein Random-Token, um seine Position auf dem Ring zu ermitteln. Beim joinen in ein Cluster, wird der Node dessen Config-File lesen. Bei einem Ausfall eines Teilsystems, wird nun aber kein Re-Balancing vorgenommen, da angenommen wird, dass es wieder online kommen wird. Um vorzubeugen, dass nicht erreichbare Nodes anderen Instanzen beitreten, wird beim Kommunizieren der Cluster-Name der Cassandra-Instanz mitgeschickt, um seinen Quellort zu bestimmen. Das hinzufügen oder entfernen eines Knoten aufgrund eines manuellen Fehlers im Konfiguration-File muss ebenfalls manuell über ein Command Line Tool erfolgen.

6. **Wie werden die Daten lokal auf den einzelnen Knoten gespeichert?** Eine Schreiboperation schreiben in ein Commit-Log. Auf jeder Maschine befindet sich eine Disk welche nur dem Commit-Log zugeteilt ist (dedicated). Das Write-Log ist dabei sequentiell. Konnten die Daten erfolgreich in's Commit-Log geschreiben werden, werden die Daten zusätzlich in eine In-Memory Struktur geladen. Ab einer gewissen Speichergrösse, wird die In-Memory-Datenstruktur auf eine Disk geschrieben. Alle Schreiboperationen sind dabei sequentiell und die Daten werden gleichzeitig für einen schnelleren Zugriff indexiert. Mit der Zeit entstehen somit mehrere Files mit den entsprechenden Dump Daten. Ein Merge Prozess fügt die verschiedenen Dump-Files wieder zu einer Datei zusammen.

   Ein Query auf den Daten arbeitet Primär auf den In-Daten-Memory. Ein Lookup auf den Files geht von neueren Files über zu den älteren Files. Die Daten werden dabei mit Hilfe eines keys gesucht. Um zu verhindern, dass Files durchsucht werden, welche den Filter nicht enthalten wird ein Bloom Filter über die vorhandenen Keys verwendet. Eine Tabelle der Bloom Filter wird sowohl auf jedem Dump-File als auch In-Memory gehalten.

8.3 Neo4J

1. **Welches sind die wichtigsten Gründe, die zur Entwicklung Ihres NoSQL Systems führten? Welche Anforderungen erfüllen diese Systeme?**
   Neo4j gehört in die Kategorie der Graphendatenbanken und wurde ursprünglich für die Echtzeitsuche von verschlagworteten(indexierten) Dokumenten in einem Online Dokumentationssystem entwickelt. „Viele aktuelle Anwendungen müssen stark vernetzte, rekursive Daten behandeln, welche sich nur unzureichend auf relationalen Datenbanken abbilden lassen. Ein gutes Beispiel dafür sind soziale Netzwerke. Auch ist es schwierig, das exakte Schema der Daten bereits zur Entwurfszeit zu kennen, speziell im Umgang mit Benutzergesteuertem Inhalt.“ Frei übersetzt aus: http://highscalability.com/neo4j-graph-database-kicks-buttox Neo4j ist in Java implementiert und sowohl als Bibliothek in die JVM eingebettet als auch als Server-Version verfügbar. Die Datenbank nutzt einen eigenen, optimierten Persistenzmechanismus für die Speicherung und Verwaltung der Graphen. Neo4j ist als open source und kommerzielle Version verfügbar. Die Datenbank ist gut dokumentiert und verfügt über einen breiten Community Support Verfügt über Schnittstellen zu diversen Programmiersprachen, ein REST Interface ist der empfohlene Zugriffsweg. Hoch skalierbar. Das .jar ist kleiner als 500kb und hat nur eine dependency. Einfaches API.

2. **Welches „logische“ Datenmodell bietet das System? Welches API bietet das System? Wie werden Daten manipuliert und abgefragt?**
   Das Datenmodell hinter neo4j ist ein Graph Datenbank System.Dabei gibt es Knoten und Kanten welche in einem Property Graphen das System ausmachen. Das heisst, zwischen den Knoten gibt es Relationships und die Kanten und Knoten haben verschiedene Properties : Um die Daten zu erhalten, spielt die Traversierung

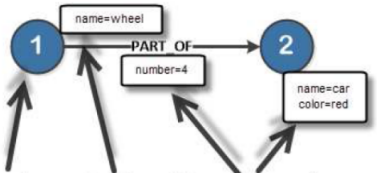

Abbildung 8.1: figure

eine wesentliche Rolle. Jeder Knoten und jede Kante speichert einen “Mini-Index” mit den eigenen verbundenen Objekten. Globale Indizes werden als Startpunkte für die Traversierung festgelegt. Weiter werden Indizes benötigt um schnell Knoten mit bestimmten Werten zu erhalten. Danach wird nach bestimmten Mustern im Graphen gesucht.

API:

API:

- Neo4j's Java API (Knoten und Beziehungen als Java-Objekte)
- REST interface (JSON-Objekte)
- DSL language
  - Abfrage Sprache Cypher ist eine deklarative Graph Query Sprache von SQL inspiriert. (ASCII)

```
(m:Movie {title: "The Matrix"})
<-[:ACTED_IN {role:"Neo"}]-
(a:Actor {name:"Keanu Reeves"})
```

Quelle: Neo4jDE_ebook.pdf

  - Gremlin: Verwendet Groovy und Pipes um Graphen zu traversieren.

Abbildung 8.2: figure

3. **Wie werden Daten über mehrere Knoten verteilt (Verteilungsarchitektur)?**

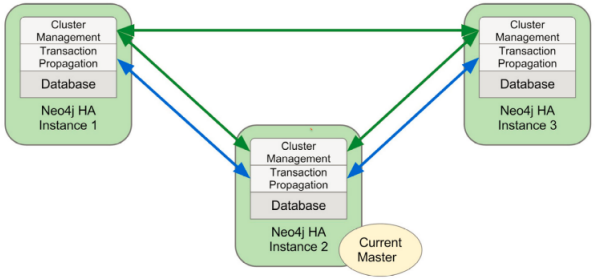

Abbildung 8.3: figure

4. **Wie werden Daten repliziert? Wie wird die Konsistenz der Daten sichergestellt?**
   Replikationen sind nur mit Neo4j Enterprise möglich. Dabei werden Cluster eingesetzt, die aus einem Master und mehreren Slaves bestehen. Die Daten werden automatisch auf die Slaves repliziert. Es sollte dabei aber darauf geachtet werden, dass Schreibvorgänge nur auf dem Master passieren. Die Replikation kann auf zwei Arten geschehen:
   - Automatisch durch Abruf der Schreibtransaktionen seitens des Slaves auf dem Master (Millisekunden - paar Minuten)
   - Automatisch durch Propagieren der Schreibtransaktionen seitens des Master auf eine vorgegebene Anzahl an Slaves (sofort). Dies wird über den push-factor definiert. Jeder Slave besitzt die ganze Datenbank. Die Skalierbarkeit ist somit nahezu linear (Read-Scaling), da Lesevorgänge beliebig auf die Slaves verteilt werden können. Bei Graphen Datenbanken ist nie die ganze Datenbank im Arbeitsspeicher, sondern nur der Teil, welcher bei der letzten Abfrage geladen wurde. Dies bietet nun die Möglichkeit über ein intelligentes und konsistentes Routing dieselben Klassen von Anfragen auf die gleichen Empfänger weiterzuleiten. So müssen nicht nach jeder Anfrage neue Teile geladen werden (Cache Sharding).

5. **Welche Vorkehrungen werden für die Verfügbarkeit und Ausfallsicherheit getroffen?**
   Dies ist nur bei einem Cluster möglich. Es sind zwei Arten des Ausfalls möglich. Fällt ein Slave aus, wird dies von den anderen Datenbankinstanzen im Netzwerk erkannt und er wird als vorübergehend nicht verfügbar gekennzeichnet. Sobald er wieder verfügbar ist, sorgt er selbst dafür, dass er wieder auf dem aktuellen Stand des Masters ist.

   Fällt der Master aus, sieht das ganze etwas anders aus. Der Cluster entscheidet, welcher Slave zum neuen Master wird. Bedingung ist lediglich, dass der Slave zum Master werden darf. Somit ist ein Failover kein Problem. Sobald der Master nicht mehr verfügbar ist, werden alle laufenden Schreibtransaktionen zurück gerollt. Der Wechsel vom Slave zum Master geschieht dann im Normalfall innert Sekunden. Während dieser Zeit sind keine Schreibtransaktionen möglich. Die einzige Ausnahme ist, wenn auf dem alten Master noch Schreibvorgänge vorhanden sind, die noch auf keine andere Datenbankinstanz repliziert werden konnten und bereits Änderungen am neuen Master gemacht wurden. In diesem Fall wird der alte Master zwei „Branches“ führen. Der erste „Branch“ ist seine alte Datenbank. Dieser „Branch“ wird abgezogen, da er nicht mehr aktuell ist, jedoch noch nicht replizierte Änderungen enthält. Dann lädt er sich für den zweiten „Branch“ eine komplette Kopie des aktuellen Zustands herunter und wird dann selbst zum Slave.

6. **Wie werden die Daten lokal auf den einzelnen Knoten gespeichert?**Nodes, Properties und Relationships werden separat voneinander gespeichert. (*.nodestore.db, *.relationshipstore.db, *.propertystore.db)

8.4 Bigtable

1. **Welches sind die wichtigsten Gründe, die zur Entwicklung Ihres NoSQL Systems führten?**

Google brauchte ein unterliegendes System für Ihre Dienste. Diese Dienste arbeiten mit grossen Datenmengen (Petabytes), sind auf viele tausende Server im Cluster verteilt und generieren viele neue Einträge, aber wenige Änderungen an vorhandenen Einträgen. Eingesetzt u. a. für folgende Produkte: Google Suche (web indexing) Google Earth Google Finance

Welche Anforderungen erfüllen diese Systeme?
breite Anwendungspalette
Skalierbarkeit
Hochverfügbarkeit
Hochleitungsfähigkeit

2. **Speicherung und Verteilungsarchitektur:**

Speicherung im Dateisystem: A Bigtable is a sparse, distributed, persistent multidimensional sorted map. Der Schlüssel für den Index besteht aus:
row key
column key
timestamp
Jeder Wert ist ein Array aus beliebigen Bytes (String).
(row:string, column:string, time:int64) -> string

Speicherung der Tabellen: Tabellen werden in „tablets“ unterteilt und so gespeichert. Die Grösse eines solchen Segments ist etwa 200 MB. Dies ist optimiert für das Google File System (GFS). Wenn viele Daten vorhanden sind werden diese komprimiert. Es wird das „SSTable file format“ verwendet.

Zugriff auf die Daten Pro Tabelle existiert ein Index, der komplett(?) in den Speicher geladen wird. Das auffinden der Daten im Dateisystem ist als Binär-Suche möglich. Der Index bezieht sich direkt auf den Block auf der HDD, so dass Zugriffe auf HDD optimiert werden können. Die Daten sind grundsätzlich unveränderbar (SSTable ist immutable). Somit ist Zugriff immer Thread-Safe. Es werden Serien von SSTables verwendet, so dass Änderungen möglich sind. Änderungen kommen aber zuerst ins änderbare „memtable“. Darin sind also immer die neuesten Änderungen gespeichert. Zugriff ist hier immer noch parallel möglich dank Copy-On-Write. Ein Tablet-Server verwaltet etwa 10-1000 Tablets.

Garbage Collection Das GFS hat eine GC (Mark-And-Sweep). Der Master-Server muss sich darum kümmern.

Verteilung der Daten: Speziell ist hier dass das GFS bereits verteilt ist. BigTable benötigt ein Cluster-Management-System zum Scheduling, Resourcen-Management und für Erkennung und Verarbeitung von Fehlern. Es gibt einen Master-Server und mehrere Tablet-Server. Solche Tablet-Servers können dynamisch entfernt oder hinzugefügt werden. Chubby wird verwendet damit Tablet-Server sich im Cluster anmelden können.

Datenmodell: persistent multidimensional sorted map.
Zeilen werden lexographisch sortiert. (Row Keys 64KB)
Tabelle wird unterteilt in Tablets. Mehrere Rows ergeben ein Tablet. Diese werden auf verschiedene Server verteilt.
Keine Limitierung in der Anzahl Spalten

Gruppierung der Spalten -> Column Families
Auf den Column Families werden die Access Control Rules definiert
Lesbarer Name
best Practice: wenige Column Families. Keine Änderungen an diesen.
Eine Gruppe -> Ein file

Timestamps Jede Zelle kann mehrer Versionen der selben Daten beinhalten. Daher wird jeder Eintrag mit einem 64bit timestamp versehen.

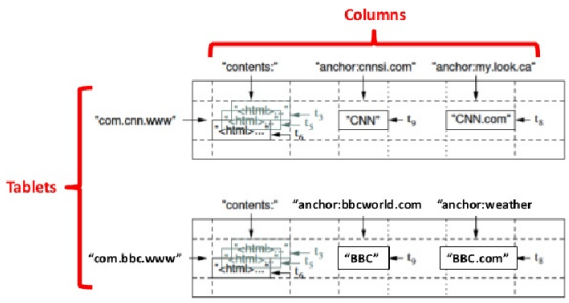

Abbildung 8.4: figure

API
Read Operations: Row by Key, Limitation of Column Families + TimeStamps, Column iterators
Write for Rows: create, delete values of column

Write for Columns Familiies: create, delete
Administration: cluster, table, column family metadata –> Access Control
jede read/write operation in einer Zeile ist atomar

3. **Wie werden Daten repliziert? Wie wird die Konsistenz der Daten sichergestellt?** BigTable verwendet ein Cluster-Management-System (Chubby) um mit verteilten Knoten zu interagieren. Jede Chubby Instanz besitzt einen Cluster von 5 aktiven Kopien, von welchem jeweils einer der Master ist und Anfragen bearbeitet. Um die Kopien konsistent zu halten wird der Paxos Algorithmus verwendet. Die gesplitteten Tablets werden auf die verschiedenen Konten verteilt.

   Chubby ist ein verstreuter Lock-Service, der ein BigTable Cluster mit mehrere tausenden Knoten koordiniert und 5 Replikationen hat. Der Klient verbindet sich jeweils zu Chubby um mit den Daten zu arbeiten. Falls für eine bestimmte Zeit Chubby unerreichbar wird, wird BigTable auch unerreichbar.

   Für Bigtable sind zurzeit zwei Replikationsmechanismen verfügbar. Einerseits die Master/Slave Replikation, bei der ein Datacenter als Master ausgewählt wird, und sämtliche Schreibzugriffe über dieses Center abgewickelt werden. Hierbei werden die Änderungen asynchron auf andere Datacenter repliziert.

   Andererseits wird der High Replication Datastore angeboten, der mittels des sog. axos-Algorithmus die Schreibanfragen auf verschiedene Datacenter verteilt und diese dann synchron auf andere Datacenter repliziert. Bigtable besitzt eigene Algorithmen, die eine automatische Lastanpassung (Skalierung) durchführen, daher ist kein manueller Eingriff notwendig.

   HBase selbst kennt keine Replikation. Dort wird mit Hilfe von Hadoop die Replikation durchgeführt.

   Verfügbarkeit Jeder TabletServer schreibt seine CommitLogs durch 2 Threads, jeder Thread schreibt ins eigene LogFile. Zu 1 Zeitpunkt ist nur 1 Thread aktiv. Wenn Writes durch den aktiven Thread zu langsam werden, wird der andere Thread als der aktive gesetzt. Alle Commits auf der Queue, die auf Write beim langsamen Thread gewartet haben, werden vom neuen Thread übernommen. · Performance: Caching! The Scan Cache is a higher-level cache that caches the key-value pairs returned by the SST able Interface to the tablet server code (bestens geeignet fürs Lesen der gleichen Daten). The Block Cache is a lower-level cache that caches SST ables blocks that were read from GFS (bestens geeignet für Lesen der Daten, die in der Nähe der neulich gelesenen Daten sind, d.h. sequentielles Lesen oder Lesen im gleichen Block).

   Ausfallsicherheit Wenn ein Tablet Server stirbt: Seine Tablets werden auf die andere Server verteilt. Um den State jedes Tablets nachzubauen, muss der CommitLog durchgesucht werden auf betreffende Commits, um diese auf den Tablets auszuführen. D.h. 100 Server – 100 Reads auf 1 File. Da es schlechtes Vorgehen ist: Der CommitLog wird sortiert nach Table-Key. CommitLog wird zuerst auf 64 MB grosse Blocks aufgeteilt, das Sortieren der Blöcke passiert parallel. Das sortierte Output ermöglicht blockweises sequentielles Lesen der nötigen Daten.

8.5 MongoDB

Bemerkung : Entschuldigung für CAPS LOCK.

1. **Welches sind die wichtigsten Gründe, die zur Entwicklung Ihres NoSQL Systems führten? Welche Anforderungen erfüllen diese Systeme?**

   DOKUMENTORIENTIERT: Verwendet BSON (Binary JSON), da sehr effizient - Bessere Interaktion mit modernen (objektorientierten) Programmiersprachen als Relationale Datenbanken

   FLEXIBILITÄT: Schemaloses Datenmodell.
   - Dokumente können jederzeit erweitert werden ohne dass hohe Aufwände entstehen (Einfaches hinzufügen von Feldern und Listen, kein Fehler beim Lesen von noch nicht existierenden Datenbanken).
   - Iterative Entwicklung möglich

   MÄCHTIGKEIT - Hoher Funktionsumfang
   - Komplexe Datentypen möglich (Strukturen mit dynamischen Attributen, Arrays, usw.)

   GESCHWINDIGKEIT/SKLAIERBARKEIT Schneller als SQL-Systeme durch:
   - Key-Value Prinzip: Zugriff direkt auf den Key anstatt diesen via Indexierung zu suchen - Vermeidung von Joins. Alle Daten können direkt über die aktuelle Liste im Dokument abgefragt werden - In-Memory lesen/schreiben

2. **Welches „logische“ Datenmodell bietet das System?**

   - MongoDB-Server besteht aus mehrere MongoDB Datenbanken
   - MongoDB Datenbank besteht aus einer odere mehreren Sammlungen eine Sammlung (hat einen Namen) besteht aus einem oder mehreren Dokumenten.

- Es ist möglich eine hierarchische Struktur der Sammlung zu bilden, indem ein Punkt eingefügt wird. Dies wird jedoch nicht so logisch abgebildet.
- ein Dokument besteht aus mehreren Feldern (Werte, Arrays, binär Daten, sub-Dokumente)
- Jedes Dokument in der gleichen Sammlung kann verschiedene Felder haben.
- Das Format wird BSON genannt und ist ähnlich wie JSON, jedoch in binärer Form damit es schneller Verarbeitet werden kann.

Beispiel Dokument:

```
{
  title : " MongoDB " ,
  last_editor : "172.5.123.91" ,
  last_modified : new Date ("9/23/2010") ,
  body : " MongoDB is a ..." ,
  categories : [" Database " , " NoSQL " , " Document Database "] ,
  reviewed : false
}
```

Vergleich zu SQL:
Sammlung <-> Tablle
Dokument <-> Zeile
Feld <-> Spalte
MongoDB ist schema frei
SQL: fünf Tabellen <-> MongoDB: zwei Sammlungen

Abbildung 8.5: figure

-

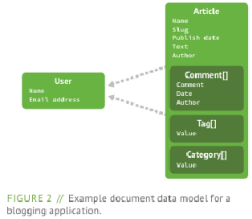

Abbildung 8.6: figure

-
-
-

3. **Wie werden Daten repliziert? Wie wird die Konsistenz der Daten sichergestellt?**

SELECT

```
db . < collection >. findOne ({ _id : 921394}) ;
db . < collection >. find ( { categories : [ " NoSQL " , " Document Databases " ] } ) ;
db . < collection >. find ( { < selection criteria >} , { < field_1 >:1 , ...} );
< field_1 >:1  die Felder
< field_1 >:0  ohne diese Felder
```

INSERT

```
db . < collection >. insert ( { title : " MongoDB " , last_editor : ... } ) ;
db . < collection >. save ( { ... } ) ; # wenn kein PK
```

UPDATE

```
db . < collection >. save ( { ... } ) ; # wenn PK
db . < collection >. update ( < criteria > , < new document > , < upsert > , < multi > ) ;
```

DELETE

```
db . < collection >. remove ( { < criteria > } ) ;
```

Abbildung 8.7: figure

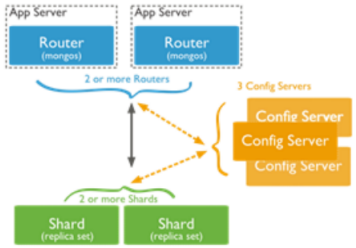

Abbildung 8.8: figure

a) MASTER-SLAVE-REPLIKATION (Alte Variante)
MongoDB unterstützt die Master-Slave-Replikation. Ein Master kann Lese- („Reads“) und Schreibzugriffe („Writes“) ausführen. Ein Slave kopiert die Daten vom Master und kann nur für Lesezugriffe oder Backups eingesetzt werden, nicht aber für Schreibzugriffe. Fällt der Masterknoten aus, so kann einer der Slaveknoten zum neuen Masterknoten umgewandelt werden (muss manuell erfolgen, Administrationsaufwand).

b) REPLICA SET (Neue Variante)
Ein Replica Set ist eine Gruppe von von Knoten, die Kopien voneinander sind. Einer dieser Gruppe wird als PrimärerKnoten gewählt. Nur über diesen werden Schreiboperationen abgewickelt. Der Primärknoten reicht diese an die anderen Knoten der Gruppe weiter. Wenn der Primärknoten ausfällt, übernimmt ein beliebiger anderer Knoten aus der Gruppe dessen Rolle (dies kann automatisch erfolgen, kein Administrationsaufwand). Vorteile der neuen Variante sind z.B: - Geringerer Administrationsaufwand

- Konsistenzbedingungen können einfach gesetzt werden (z.B. Schreiben ist erfolgreich wenn mehr als die Hälfte des Replica Sets die Änderungen übernommen hat)
- Grosse Skalierbarkeit, da nahezu jeder Knoten aus einem Replica Set für Leseoperationen verwendet werden kann

4. **Welche Vorkehrungen werden für die Verfügbarkeit und Ausfallsicherheit getroffen?**

REPLICATION Operationen welche die Datenbank auf dem PRIMARY verändern, werden in einem Log namens oplog gespeichert. Der 'oplog' enthält eine geordnete Menge von idempotenten Operationen, welche auf den SSecondariesrepliziert werden. Die Grösse des oplogs beträgt standardmässig 5des freien Festplattenspeichers. Der oplog enthält Änderungen einiger Stunden. Sollte ein Secondary ausfallen, kann er so wieder äufholen". Sollte ein Secondary länger als die Periode vom oplog beträgt ausfallen, dann werden mittels ïnitial synchronizationälle Datenbanken usw. komplett repliziert.

ELECTIONS AND FAILOVER Falls der Primary aus irgendeinem Grund ausfallen sollte, dann bestimmen die Secondaries unter sich einen neuen Primary. Dieser Prozess wird Ëlection"genannt. Sobald der neue Primary bestimmt wurde, konfigurieren sich die anderen Seconaries selber, so dass sie nun Updates vom neuen Primary erhalten. Sollte der ausgefallene Primary wieder online kommen, erkennt er, dass er nicht mehr der Primary ist und wird zu einem Secondary.

ELECTION PRIORITY Der ËlectionProzess kann mit einer Prioritäts-Konfiguration beinflusst werden. Der Secondary mit der höchsten Priorität wird dann Primary. Zum Beispiel könnte man alle Instanzen in einem SSecondary Data Centerßo konfigurieren, dass nur wenn das komplette Primary Data Centeräusfällt, einer der Secondarys dann Primary wird.

CONFIGURABLE WRITE AVAILABILITY MongoDB besitzt die Möglichkeit, dass ein Client bei jeder Abfrage einen "write concernmitteilen kann. Dieser gibt an, wann der Client eine Antwort erhält. Der Client wartet dann solange auf eine Antwort bis der gewünschte "write concernëingetreten ist und kann so z.B. sicherstellen, dass die Daten definitiv geschrieben wurden.

Folgende Optionen sind z.B. möglich:
- Client braucht keine Antwort, (One-Way) -> Keine Garantie, dass Daten geschrieben werden
- Client will eine Bestätigung sobald die Änderungen auf mehreren Replicas vollzogen worden sind.
- Client will eine Bestätigung sobald die Änderungen auf der Mehrheit der Replicas vollzogen worden sind.
- Client will eine Bestätigung sobald die Änderungen auf allen Replicas vollzogen worden sind.
- Und weitere, einstellbare, Möglichkeiten wären z.B: Mind. 2 Replicas enthalten die Änderungen im Primary-Datacenter und min. 1 Replica im Secondary Datacenter.

5. **Abspeicherung Lokal**

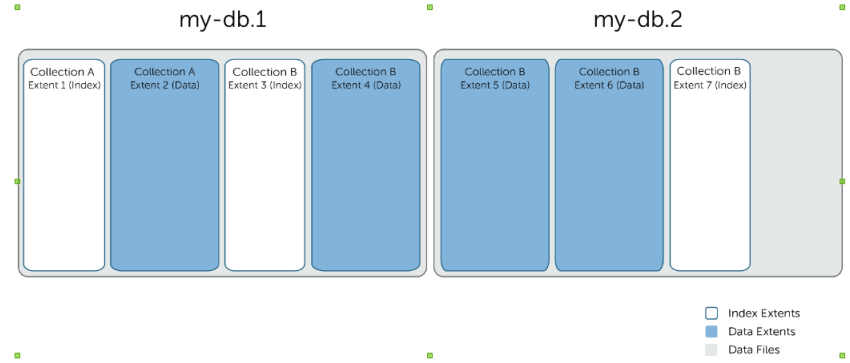

Abbildung 8.9: figure