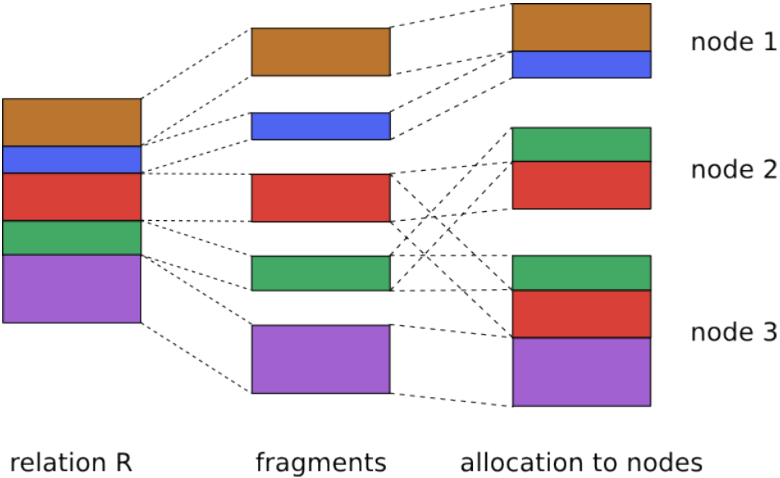
Fragmentierung

Grundlagen

Fragmentierung

- Notwendig in Shared-Nothing Architekturen (Top-Down Ansatz)
- Beeinflusst
  - Kommunikationskosten
  - Lastbalancierung
  - Verfügbarkeit
- Wichtige Aspekte
  - Fragmentierung
    Relationen werden in kleinere, disjunkte Fragmente zerlegt
  - Allokation
    Fragmente werden auf die beteiligten Rechner verteilt
  - Replikation
    Kopien einiger Fragmente werden auf mehreren Rechnern gespeichert


node 1
node 2
node 3
relation R
fragments
allocation to nodes

Wichtige Aspekte:

- Granularität der Fragmentierung
  - Wie groß soll ein Fragment sein?
  - Welche Teile der Relation sollen welchem Fragment zugewiesen werden?
- Allokation
  - Welche Fragmente sollen welchen Rechnern zugewiesen werden?
  - Welche Fragmente sollen repliziert werden, welche sollen nur einmal gespeichert werden?

Wenn jedes Fragment nur einmal gespeichert wird (also keine Replikation verwendet wird), spricht man auch von Partitionierung.

Granularität der Fragmentierung

Verteilen von Fragmenten oder Relationen an die einzelnen Rechner:

- Relation
  - Operationen auf Relationen können immer auf einem einzelnen Rechner ausgeführt werden
  - Einfachere Überwachung von Integritätsbedingungen
- Fragmente einer Relation
  - Konzentration der Daten auf jedem Rechner, die dort zugegriffen werden
  - bessere Lastbalancierung
  - Reduktion des Berechnungsaufwands, wenn Operationen nur auf einzelne Fragmente zugreifen müssen
  - Unterstützung von paralleler Anfrageausführung

Fragmentierung

Fragmentierung zerlegt eine Relation R in mehrere Fragmente $F_R := \{R_1, R_2, \ldots, R_n\}$.

Korrektheitsregeln

Eine ordnungsgemäße Fragmentierung muss folgende Korrektheitsregeln erfüllen:

- Vollständigkeit
  Die Fragmente enthalten alle Daten
- Disjunktheit
  Fragmente überlappen nicht
- Rekonstruktion
  Die Fragmentierung erhält die Daten und Eigenschaften der ursprünglichen Relation

Fragmentierung

Ziele

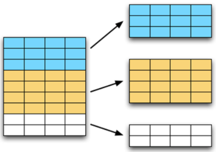
- Ausnutzen der Verarbeitungskapazität mehrerer Rechner, indem Operationen in Teiloperationen zerlegt werden, die parallel ausgeführt werden können

Anforderungen

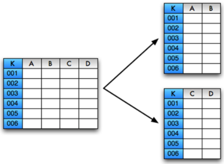
- Alle Partitionen (nicht unbedingt alle Fragmente) sollten gleich groß sein
- Partitionen sollten die Ausführung von Operationen unterstützen

Alternativen für die Fragmentierung

- Tupel oder Attribute, d.h. horizontal oder vertikal



Horizontale Fragmentierung



Vertikale Fragmentierung

Datenbankbeispiel

Employees	EID	EName	Title
	E1	Just Vorfan	Programmer
	E2	Ann Joy	Elect. Engineer
	E3	Lilo Pause	Programmer
	E4	Claire Grube	Mech. Engineer
	E5	John Doe	Syst. Analyst

Assignment	Eno	Pno	Duration
	E1	P1	5
	E2	P4	4
	E2	P1	6
	E3	P4	3
	E4	P1	4
	E4	P3	5
	E5	P2	7

Projects	Pno	Pname	Budget	Location
	P1	Database Development	200.000	Saarbrücken
	P2	Maintenance	150.000	Munich
	P3	Web Design	100.000	Paris
	P4	Customizing	250.000	Saarbrücken

Salary	Title	Salary
	Elect. Eng.	60.000
	Syst. Analyst	55.000
	Mech. Engineer	65.000
	Programmer	90.000

Universität Trier

Horizontale Fragmentierung

Horizontale Fragmentierung

- Die Relation wird *horizontal* aufgeteilt, d.h. Tupel werden auf Fragmente verteilt
- Die Fragmentierung wird durch Selektionsprädikate definiert, d.h.
  $R_i := \sigma_{P_i}(R) \quad (1 \leq i \leq n)$
- Man nennt P_i auch ein *Fragmentierungsprädikat*
- Aufteilen von PROJECTS gemäß des Attributs PNO in PROJECTS_1 und PROJECTS_2

PROJECTS_1

PNo	PName	Budget	Location
P1	Database Development	200.000	Saarbrücken
P2	Maintenance	150.000	Munich

PROJECTS_2

PNo	PName	Budget	Location
P3	Web Design	100.000	Paris
P4	Customizing	250.000	Saarbrücken

Universität Trier

Horizontale Fragmentierung

Grundlegende Methoden:

- Bereichspartitionierung
  Jedem Fragment wird ein anderer Wertebereich zugewiesen:
  - fragment_1: Budget $\geq$ 200.000
  - fragment_2: Budget $<$ 200.000
- Wertelisten
  Für jedes Fragment wird eine Liste von Attributwerten vorgegeben:
  - fragment_1: PNo $\in$ {'P1', 'P2'}
  - fragment_2: PNo $\in$ {'P3', 'P4'}
- Hashpartitionierung
  Auf jedes Tupel wird die gleiche Hashfunktion angewendet, jedem Fragment dann ein anderer Wertebereich der Hashfunktion zugewiesen:
  - fragment_1: $0 \leq h(t) \leq 50$
  - fragment_2: $50 < h(t) \leq 100$

Universität Trier

Horizontale Fragmentierung

Horizontale Vollständigkeitsregel

Jedes Tupel einer Relation R ist in mindestens einem Fragment enthalten, d.h. $R = R_1 \cup R_2 \cup R_3 \cup \cdots \cup R_n$

Horizontale Disjunktheitsregel

Jedes Tupel ist in höchstens einem Fragment enthalten, d.h.
$\forall 1 \leq i \neq k \leq n : R_i \cap R_k = \emptyset$

Horizontale Rekonstruktionsregel

Primärschlüssel müssen eindeutig bleiben und Fremdschlüssel müssen erhalten werden. Rekonstruktion: $R = \bigcup_{1 \leq i \leq n} R_i$

Horizontale Fragmentierung

- Bestimmung guter Prädikate für eine horizontale Fragmentierung
  - Manuell
  - Automatisch zur Entwurfszeit
  - Automatisch zur Laufzeit

Horizontale Fragmentierung...

- Manuelle horizontale Fragmentierung
  - Datenbankadministrator nutzt sein Wissen über Daten und ihre Verwendung
- Automatische horizontale Fragmentierung zur Entwurfszeit
  - Weit verbreitet in Firmendatenbanken
  - Nutzt Wissen über nachgefragte Daten und häufige Anfragen aus
  - Bestimmt "optimale" Fragmentierung, so dass die geschätzte Gesamtperformance maximiert wird
  - Problem:
    Was passiert, wenn sich Art und Häufigkeit der Anfragen ändern?
- Automatische horizontale Fragmentierung zur Laufzeit
  - Verwendet in Cloud-Speichersystemen
  - Das System bestimmt automatisch einen guten Fragmentierungsplan anhand von Verwendungsstatistiken
  - Keine Zusatzinformationen des Administrators notwendig
  - Problem:
    gute Fragmentierung zur Laufzeit ist schwierig

Naive automatische horizontale Fragmentierung

Einfacher Algorithmus zur automatischen horizontalen Partitionierung zur Entwurfszeit

- Input
  - Relation $R(A_1, \ldots, A_n)$, wobei A_i ein Attribut mit der Domäne $D_i = Dom(A_i)$ ist
  - Menge von Anfragen
- Output
  - Menge M von Selektionsausdrücken zur Fragmentierung

Naive automatische horizontale Fragmentierung

Algorithmus

1. Sammle Anfragen und Statistiken
2. Identifiziere einfache Prädikate in Anfragen
   - $p_j : A_i \ \theta \ Value$, mit Operator $\theta \in \{<, \leq, >, \geq, =, \neq\}$ und $Value \in Dom(A_i)$
   - p_j definiert eine mögliche binäre Partitionierung von R

Beispiel

- Gegebene Anfragen:
  - q_1: SELECT PName FROM PROJECTS WHERE PNo = 'P1'
  - q_2: SELECT Location FROM PROJECTS WHERE Budget BETWEEN 125.000 AND 225.000
- Prädikate:
  - q_1: $PNo =$ 'P1'
  - q_2: $Budget \geq$ 125.000, $Budget \leq$ 225.000

Algorithmus

3. Definition aller möglichen Minterme
   - Menge $M_n(P)$ aller n-stelligen Minterme für eine Menge P von n Prädikaten
   - Ein Minterm ist eine Konjunktion von einfachen Prädikaten
   - $M_n(P) = \{m \mid m = \bigwedge_{i=1}^{n} p_i^*, p_i \in P\}$
   - p_i^* repräsentiert das Prädikat p_i in seiner negierten ($p_i^- \equiv \neg p_i$) oder nicht-negierten Form ($p_i^+ = p_i$)
     Minterme können nicht beide Versionen eines Prädikates enthalten
   - $M_n(P)$ definiert eine vollständige und disjunkte Fragmentierung von R
     - $R = \bigcup_{m \in M_n(P)} \sigma_m(R)$
     - $\forall m_i, m_k \in M_n(P), m_i \neq m_k : \sigma_{m_i}(R) \cap \sigma_{m_k}(R) = \emptyset$

Universität Trier

Beispiel

- Gegeben sei die folgende Menge P von Prädikaten:
  - p_1 : $PNo =$ 'P1'
  - p_2 : $Budget \geq 125.000$
  - p_3 : $Budget \leq 225.000$
- Wir können die folgende Menge $M_3(P)$ von Mintermen ableiten:
  - $m_1 : p_1^+ \wedge p_2^+ \wedge p_3^+$
  - $m_2 : p_1^+ \wedge p_2^+ \wedge p_3^-$
  - $m_3 : p_1^+ \wedge p_2^- \wedge p_3^+$
  - $m_4 : p_1^+ \wedge p_2^- \wedge p_3^-$
  - $m_5 : p_1^- \wedge p_2^+ \wedge p_3^+$
  - $m_6 : p_1^- \wedge p_2^+ \wedge p_3^-$
  - $m_7 : p_1^- \wedge p_2^- \wedge p_3^+$
  - $m_8 : p_1^- \wedge p_2^- \wedge p_3^-$

Beispiel

- Gegeben sei die folgende Menge P von Prädikaten:
  - p_1 : $PNo =$ 'P1'
  - p_2 : $Budget \geq 125.000$
  - p_3 : $Budget \leq 225.000$
- Wir können die folgende Menge $M_3(P)$ von Mintermen ableiten:
  - m_1 : $PNo =$ 'P1' $\wedge$ $Budget \geq 125.000 \wedge Budget \leq 225.000$
  - m_2 : $PNo =$ 'P1' $\wedge$ $Budget \geq 125.000 \wedge \neg(Budget \leq 225.000)$
  - m_3 : $PNo =$ 'P1' $\wedge$ $\neg(Budget \geq 125.000) \wedge Budget \leq 225.000$
  - m_4 : $PNo =$ 'P1' $\wedge$ $\neg(Budget \geq 125.000) \wedge \neg(Budget \leq 225.000)$
  - m_5 : $\neg(PNo =$ 'P1'$)$ $\wedge$ $Budget \geq 125.000 \wedge Budget \leq 225.000$
  - m_6 : $\neg(PNo =$ 'P1'$)$ $\wedge$ $Budget \geq 125.000 \wedge \neg(Budget \leq 225.000)$
  - m_7 : $\neg(PNo =$ 'P1'$)$ $\wedge$ $\neg(Budget \geq 125.000) \wedge Budget \leq 225.000$
  - m_8 : $\neg(PNo =$ 'P1'$)$ $\wedge$ $\neg(Budget \geq 125.000) \wedge \neg(Budget \leq 225.000)$

Algorithmus

4. Eliminierung nichterfüllbarer Minterme

Beispiel

- Gegeben sei die folgende Menge von Minterms:
  - m_1 : $PNo =$ 'P1' $\wedge$ $Budget \geq 125.000 \wedge Budget \leq 225.000$
  - m_2 : $PNo =$ 'P1' $\wedge$ $Budget \geq 125.000 \wedge \neg(Budget \leq 225.000)$
  - m_3 : $PNo =$ 'P1' $\wedge \neg(Budget \geq 125.000) \wedge Budget \leq 225.000$
  - $\cancel{m_4 : PNo = \text{'P1'} \wedge \neg(Budget \geq 125.000) \wedge \neg(Budget \leq 225.000)}$
  - m_5 : $\neg(PNo =$ 'P1'$) \wedge Budget \geq 125.000 \wedge Budget \leq 225.000$
  - m_6 : $\neg(PNo =$ 'P1'$) \wedge Budget \geq 125.000 \wedge \neg(Budget \leq 225.000)$
  - m_7 : $\neg(PNo =$ 'P1'$) \wedge \neg(Budget \geq 125.000) \wedge Budget \leq 225.000$
  - $\cancel{m_8 : \neg(PNo = \text{'P1'}) \wedge \neg(Budget \geq 125.000) \wedge \neg(Budget \leq 225.000)}$

Universität Trier

Algorithmus

5. Eliminierung von abhängigen Prädikaten
   Eliminiere Prädikate in einem Minterm, die abhängig sind (Implikationen und funktionale Abhängigkeiten)

Die resultierenden Terme sind technisch keine Minterme mehr (weil sie nicht alle Prädikate enthalten), wir behalten aber den Begriff bei.
Beispiel

- Gegeben sei die folgende Menge von Mintermen:
  - $m_1 : PNo = \text{'P1'} \wedge Budget \geq 125.000 \wedge Budget \leq 225.000$
  - $\cancel{m_2 : PNo = \text{'P1'} \wedge Budget \geq 125.000 \wedge \neg(Budget \leq 225.000)}$
  - $m_2 : PNo = \text{'P1'} \wedge Budget > 225.000$
  - $\cancel{m_3 : PNo = \text{'P1'} \wedge \neg(Budget \geq 125.000) \wedge Budget \leq 225.000}$
  - $m_3 : PNo = \text{'P1'} \wedge Budget < 125.000$
  - $m_5 : \neg(PNo = \text{'P1'}) \wedge Budget \geq 125.000 \wedge Budget \leq 225.000$
  - $\cancel{m_6 : \neg(PNo = \text{'P1'}) \wedge Budget \geq 125.000 \wedge \neg(Budget \leq 225.000)}$
  - $m_6 : \neg(PNo = \text{'P1'}) \wedge Budget > 225.000$
  - $\cancel{m_7 : \neg(PNo = \text{'P1'}) \wedge \neg(Budget \geq 125.000) \wedge Budget \leq 225.000}$
  - $m_7 : \neg(PNo = \text{'P1'}) \wedge Budget < 125.000$

Algorithmus

6. Schätze die Selektivität jedes Minterms m_i: $sel(m_i) = \frac{|\sigma_{m_i}(R)|}{card(R)}$ wobei $card(R)$ die Zahl von Tupeln in Relation R angibt, entferne Minterme mit Selektivität 0

Beispiel

- Gegeben sei die folgende Menge von Mintermen:
  - m_1 : $PNo =$ 'P1' $\wedge$ $Budget \geq 125.000 \wedge Budget \leq 225.000$, $sel(m_1) = \frac{1}{4} = 0.25$
  - m_2 : $PNo =$ 'P1' $\wedge$ $Budget > 225.000$, $sel(m_2) = \frac{0}{4} = 0$
  - ~~m_2 : $PNo =$ 'P1' $\wedge$ $Budget > 225.000$, $sel(m_2) = \frac{0}{4} = 0$~~
  - m_3 : $PNo =$ 'P1' $\wedge$ $Budget < 125.000$, $sel(m_3) = \frac{0}{4} = 0$
  - ~~m_3 : $PNo =$ 'P1' $\wedge$ $Budget < 125.000$, $sel(m_3) = \frac{0}{4} = 0$~~
  - m_5 : $\neg(PNo =$ 'P1'$)$ $\wedge$ $Budget \geq 125.000 \wedge Budget \leq 225.000$ $sel(m_1) = \frac{1}{4} = 0.25$
  - m_6 : $\neg(PNo =$ 'P1'$)$ $\wedge$ $Budget > 225.000$, $sel(m_1) = \frac{1}{4} = 0.25$
  - m_7 : $\neg(PNo =$ 'P1'$)$ $\wedge$ $Budget < 125.000$, $sel(m_1) = \frac{1}{4} = 0.25$

PROJECTS

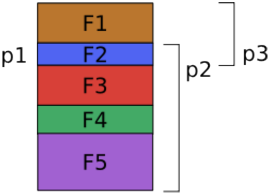
PNo	PName	Budget	Location
P1	Database Development	200.000	Saarbr.
P2	Maintenance	150.000	Munich
P3	Web Design	100.000	Paris
P4	Customizing	250.000	Saarbr.

Universität Trier

Algorithmus

7. Bestimme minimale und vollständige Mengen von Mintermen für die Definition von Fragmenten
   - Minimal: Für jedes Paar von Fragmenten gibt es mindestens eine Anfrage, die auf beide Fragmente unterschiedlich zugreift
   - Vollständig: Die Zugriffswahrscheinlichkeit für jedes Tupel innerhalb eines Fragments ist ähnlich

Beispiel
Alternative Relation PROJECT *mit mehr Tupeln*



Gegebene Prädikate

- $p_1 : PNo =$ 'P1', $p_2 : Budget \geq 125.000$, $p_3 : Budget \leq 225.000$

Gegebener Minterm

- $m_1 : PNo =$ 'P1' $\land\ Budget \geq 125.000 \land Budget \leq 225.000$
  $(p_1^+ \land p_2^+ \land p_3^+)$

Fragmentierung gemäß m_1

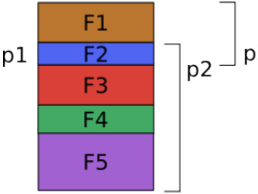
- $R_1 = \{F_2\}$

Visualisierung der Fragmentierung mit Mintermen

Beispiel
Alternative Relation PROJECT *mit mehr Tupeln*

Alle möglichen Minterme und ihre Fragmente außer $(p_1^- \wedge p_2^- \wedge p_3^-)$

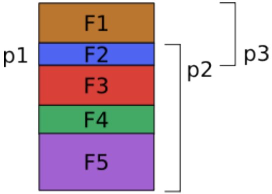
- $m_1 : PNo = \text{'P1'} \wedge Budget \geq 125.000 \wedge Budget \leq 225.000$, $R_1 = \{F_2\}$
- $m_2 : PNo = \text{'P1'} \wedge Budget > 225.000$, $R_2 = \{\}$
- $m_3 : PNo = \text{'P1'} \wedge Budget < 125.000$, $R_3 = \{\}$
- $m_5 : \neg(PNo = \text{'P1'}) \wedge Budget \geq 125.000 \wedge Budget \leq 225.000$, $R_5 = \{\}$
- $m_6 : \neg(PNo = \text{'P1'}) \wedge Budget > 225.000$, $R_6 = \{F_3, F_4, F_5\}$
- $m_7 : \neg(PNo = \text{'P1'}) \wedge Budget < 125.000$, $R_7 = \{F_1\}$

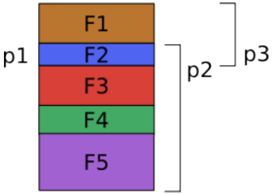


Beispiel
Alternative Relation PROJECT *mit mehr Tupeln*

Resultierende Fragmentierung

- m_1 : $PNo =$ 'P1' $\wedge$ $Budget \geq 125.000 \wedge Budget \leq 225.000$, $R_1 = \{F_2\}$
- m_6 : $\neg(PNo =$ 'P1'$)$ $\wedge$ $Budget > 225.000$, $R_6 = \{F_3, F_4, F_5\}$
- m_7 : $\neg(PNo =$ 'P1'$)$ $\wedge$ $Budget < 125.000$, $R_7 = \{F_1\}$





Diese drei Prädikate können drei Fragmente definieren

- p_2 und p_3 können zusammen drei Fragmente definieren
- p_1, p_2, und p_3 können zusammen ebenfalls nur drei Fragmente definieren

Die Verwendung von zwei der drei Prädikate genügt
Optimierung: Minimiere die Zahl der Prädikate

Naive automatische horizontale Fragmentierung

Zusammenfassung

- Der naive Algorithmus generiert alle möglichen Fragmentierungen und behält die beste
- 2^n mögliche Fragmentierungen, wobei n die Anzahl der Mintermprädikate ist
- Eine Teilmenge der Prädikate kann zur gleichen Fragmentierung führen wie alle Prädikate

Optimale horizontale Fragmentierung

- Ziel
  - Bestimmen einer vollständigen, nichtredundanten, und minimalen horizontalen Fragmentierung einer Relation R und einer gegebenen Menge P von Prädikaten
- Eingabe:
  - P ist die Menge von Prädikaten über R
  - $M(P)$ ist die Menge von **praktisch relevanten** Mintermen
  - $F(P)$ ist die Menge von Fragmenten, die aus der Menge von Mintermen über R hervorgehen:
    $F(P) = \{R_1, R_2, \ldots, R_n\}$, $R_i(m_i) := \sigma_{m_i}(R)$ with $m_i \in M(P)$ und σ_{m_i} ist das zu Minterm m_i gehörende Selektionsprädikat
- Ausgabe:
  - Die Menge Q von Prädikaten, die eine vollständige, nichtredundante, und minimale Fragmentierung der Relation R definiert

Vollständigkeit von Prädikatmengen

Eine Menge P von einfachen Prädikaten ist *vollständig*, wenn jede Anfrage mit gleicher Wahrscheinlichkeit auf ein Tupel eines beliebigen Mintermfragments zugreift, das gemäß P definiert wurde.

Beispiel
$P = \{Location =$ 'Paris', $Location =$ 'Saarbrücken', $Location =$ 'Munich'$\}$

- Vollständig, wenn Anfragen die Relation PROJECT nur basierend auf Location zugreifen
- Unvollständig, wenn es auch Anfragen gibt, die auf die Relation PROJECT zugreifen basierend auf der Bedingung, dass das Budget kleiner als 125.000 ist; in diesem Fall wird auf einige Tupel innerhalb des gleichen Fragments häufiger zugegriffen als auf andere
  In diesem Fall wäre sie vollständig, wenn z.B.
  $P = \{Budget < 125.000, Location =$ 'Paris', $Location =$ 'Saarbrücken', $Location =$ 'Munich'$\}$

Universität Trier

Minimalität der Prädikatmengen

Eine Menge P von einfachen Prädikaten ist *minimal*, wenn alle ihre Prädikate *relevant* sind.

Relevanz eines Prädikats

Ein Prädikat ist *relevant*, wenn es die Fragmentierung beeinflusst und Anfragen auf die jeweils erzeugte Fragmentierung mit und ohne das Prädikat unterschiedlich zugreifen.

Fragmentierungen, die aus vollständigen und minimalen Prädikaten/Mintermen abgeleitet werden, heißen vollständig und minimal.

Relevanz eines Prädikats

Ein Prädikat ist *relevant*, wenn es die Fragmentierung beeinflusst und Anfragen auf die jeweils erzeugte Fragmentierung mit und ohne das Prädikat unterschiedlich zugreifen.

Beispiel

- Das Prädikat p_i führt dazu, dass Fragment F weiter in Fragmente F_{i+} und F_{i-} aufgeteilt wird.
- Es ist relevant, wenn es eine Anfrage gibt, die auf F_{i+} und F_{i-} unterschiedlich zugreift.

Formal:

- Gegeben seien Minterme m_{i+} und m_{i-}, wobei m_{i+} p_i^+ enthalte und m_{i-} p_i^- enthalte
- Zugriff: $\mathsf{acc}(m)$, Fragmentgröße: $\mathsf{card}(F)$
- p_i ist relevant, wenn $\frac{\mathsf{acc}(m_{i+})}{\mathsf{card}(F_{i+})} \neq \frac{\mathsf{acc}(m_{i-})}{\mathsf{card}(F_{i-})}$

Beispiel 3.1 (Relevanz eines Prädikats)

Wir nehmen an, dass 300 Anfragen auf Fragment F zugreifen, das Größe 3000 hat. Nachdem p_i zu den Prädikaten hinzugefügt wurde, wird F in zwei Fragmente F_{i+} (mit Größe 300) und F_{i-} (mit Größe 2700) zerlegt. Wir nehmen weiter an, dass 100 Anfragen nur auf F_{i+} zugreifen, während die übrigen 200 Anfragen auf beide Fragmente zugreifen. Wir können nun die beiden Brüche berechnen:

Fragment F_{i+}: $\frac{300}{300} = 1$
Fragment F_{i-}: $\frac{200}{2700} = 0.074$

Offensichtlich ist der Zugriff auf beide Fragmente deutlich unterschiedlich, also ist p_i relevant.

Optimale horizontale Fragmentierung

- Ziel
  - Bestimmen einer vollständigen, nichtredundanten, und minimalen horizontalen Fragmentierung einer Relation R und einer gegebenen Menge P von Prädikaten
- Eingabe:
  - P ist die Menge von Prädikaten über R
  - $M(P)$ ist die Menge von **praktisch relevanten** Mintermen
  - $F(P)$ ist die Menge von Fragmenten, die aus der Menge von Mintermen über R hervorgehen:
    $F(P) = \{R_1, R_2, \ldots, R_n\}$, $R_i(m_i) := \sigma_{m_i}(R)$ with $m_i \in M(P)$ und σ_{m_i} ist das zu Minterm m_i gehörende Selektionsprädikat
- Ausgabe:
  - Die Menge Q von Prädikaten, die eine vollständige, nichtredundante, und minimale Fragmentierung der Relation R definiert

Universität Trier

Praktisch relevante Minterme

- Eliminierung unerfüllbarer Minterme
  Wenn sich zwei Terme p_i^* und p_j^* von Minterm $m \in M(P)$ widersprechen, dann ist m unerfüllbar und kann aus $M(P)$ entfernt werden.
- Eliminierung von abhängigen Prädikaten
  Wenn Term p_i^* von $m \in M(P)$ einen anderen Term p_j^* von m impliziert (durch eine funktionale Abhängigkeit), dann kann p_j^* entfernt werden.

$Q := \emptyset$
$M(Q) := \{true\}$
forall $p \in P$ **do**
    $Q' := Q \cup \{p\}$
    bestimme $M(Q')$ und $F(Q')$
    vergleiche $F(Q')$ und $F(Q)$
    if $F(Q')$ *klare Verbesserung* im Vergleich zu $F(Q)$ **then**
        $Q := Q'$
        forall $q \in Q \setminus \{p\}$ **do** /* *unnötige Fragmentierung?* */
            $Q' := Q \setminus \{q\}$
            bestimme $M(Q')$ und $F(Q')$
            vergleiche $F(Q')$ und $F(Q)$
            if $F(Q)$ keine *klare Verbesserung* im Vergleich zu $F(Q')$ **then**
                $Q := Q'$ /* *entferne q aus Q* */
            end
        end
    end

Abgeleitete horizontale Fragmentierung

Abgeleitete horizontale Fragmentierung

- Primäre horizontale Fragmentierung
  - Aufteilen einer einzigen Relation R
  - Verwenden von Fragmentierungsausdrücken und -kriterien, die sich nur auf R beziehen
  - Keine Abhängigkeiten von anderen Relationen
- Abgeleitete horizontale Fragmentierung
  - Aufteilen einer Relation *in Abhängigkeit von* einer anderen Relation

Wir betrachten *mehrere* Relationen

EMPLOYEES

<u>EID</u>	EName	Title

ASSIGNMENT

<u>ENo</u>	<u>PNo</u>	Duration

PROJECTS

<u>PNo</u>	PName	Budget	Location

SALARY

<u>Title</u>	Salary

Fremdschlüsselbeziehungen

- EMPLOYEES.EID $\mapsto$ ASSIGNMENT.ENo
- PROJECTS.PNo $\mapsto$ ASSIGNMENT.PNo
- SALARY.Title $\mapsto$ EMPLOYEES.Title

Abgeleitete horizontale Fragmentierung

Ziel

- Horizontale Fragmentierung einer Relation S basierend auf der Fragmentierung einer anderen Relation R

Gegeben sind

- Relationen R und S
- $R.A$ ist Fremdschlüssel in S
- R wurde bereits fragmentiert in $R_1, R_2, \ldots, R_n$

Ergebnis

- Relation S ist partitioniert unter Berücksichtigung der Fragmente von R

Abgeleitete horizontale Fragmentierung

Fragmentierung von S basierend auf der Fragmentierung von R (R_1, R_2, ..., R_n)

- Verwendung des Semijoin-Operators
- $S_i = S \ltimes R_i = S \ltimes \sigma_{P_i}(R) = \pi_{S.*}(S \bowtie \sigma_{P_i}(R))$
- Fragmentierungsausdruck σ_{P_i} bezieht sich nur auf R
- Vollständigkeit
  Wenn der Semijoin verlustfrei ist, d.h. wenn es im Fremdschlüsselattribut von S keine Nullwerte gibt
- Disjunktheit
  Folgt aus der Disjunktheit der Fragmente von R
- Rekonstruierbarkeit
  $S = \bigcup_{1 \leq i \leq n} S_i$

Universität Trier

Abgeleitete horizontale Fragmentierung

Gegeben die Fragmente PROJECT$_1$ und PROJECT$_2$ teilen wir die Relation ASSIGNMENT wiefolgt auf:

PROJECTS$_1$

PNo	PName	Budget	Location
P1	Database Development	200.000	Saarbr.
P2	Maintenance	150.000	Munich

PROJECTS$_2$

PNo	PName	Budget	Location
P3	Web Design	100.000	Paris
P4	Customizing	250.000	Saarbr.

ASSIGNMENT

ENo	PNo	Duration
E1	P1	5
E2	P4	4
E2	P1	6
E3	P4	3
E4	P1	4
E4	P3	5
E5	P2	7

$$\text{ASSIGNMENT}_1 = \text{ASSIGNMENT} \ltimes \text{PROJECTS}_1$$
$$\text{ASSIGNMENT}_2 = \text{ASSIGNMENT} \ltimes \text{PROJECTS}_2$$

ASSIGNMENT$_1$

ENo	PNo	Duration
E1	P1	5
E2	P1	6
E4	P1	4
E5	P2	7

ASSIGNMENT$_2$

ENo	PNo	Duration
E2	P4	4
E3	P4	3
E4	P3	5

Universität Trier

Erwünschte Eigenschaften einer Fragmentierung

Fragmentierung mit guter Joincharakteristik

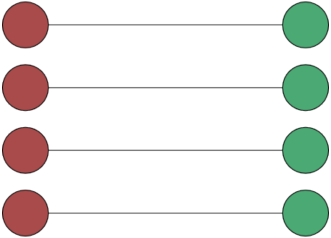
- Die Performance von Joins in einem verteilten Datenbanksystem wird gesteigert, wenn
  - die Relationen/Fragmente, die am Join teilnehmen, klein sind
  - Joins auf einem einzelnen Rechner ausgeführt werden
  - häufige Anfragen so effizient wie möglich ausgeführt werden
    Minimale Anzahl von Fragmenten, die gejoint werden müssen

Informeller Ansatz, um die Joinperformance zu schätzen

- Fragment-Join-Graphen
  - Jedes Fragment ist ein Knoten
  - Wenn ein Join zwischen zwei Fragmenten ein nichtleeres Ergebnis erzielen kann, werden die beiden zugehörigen Knoten verbunden
  - Je weniger Kanten, desto besser ist die Fragmentierung

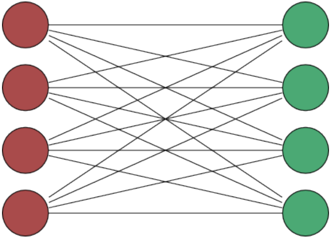
Erwünschte Eigenschaften einer Fragmentierung

Gut: einfacher Joingraph

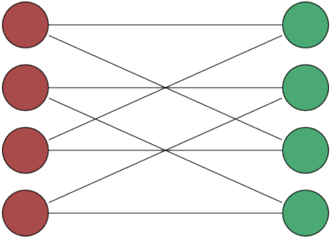


Erwünschte Eigenschaften einer Fragmentierung

Schlecht: vollständiger Joingraph



Okay: Joingraph mit wenigen Kreuzkanten



Vertikale Fragmentierung

Vertikale Fragmentierung

- Vertikales Aufteilen von Relationen bedeutet, dass Attribute zwischen den Fragmenten verteilt werden
- Attribute des Primärschlüssels werden in allen Fragmenten repliziert
- Beispiel: Aufteilen von PROJECTS in zwei Fragmente PROJECTS_1 und PROJECTS_2 – *PNo ist der Primärschlüssel*

PROJECTS_1

PNo	PName	Location
P1	Database Development	Saarbr.
P2	Maintenance	Munich
P3	Web Design	Paris
P4	Customizing	Saarbr.

PROJECTS_2

PNo	Budget
P1	200.000
P2	150.000
P3	100.000
P4	250.000

Vertikale Vollständigkeitsregel

Jedes Attribut ist in einem der Fragmente enthalten

Vertikale Disjunktheitsregel

Jedes Nicht-Primärschlüsselattribut ist in höchstens einem Fragment enthalten, während die Primärschlüsselattribute in allen Fragmenten enthalten sind (eingeschränkte Disjunktheit), d.h. $R_i := \pi_{K,A_{i_1},\ldots,A_{i_m}}(R)$

Vertikale Rekonstruierbarkeitsregel

Die ursprüngliche Relation kann durch einen Natural Join wiederhergestellt werden, d.h. $R = R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n$

Universität Trier

Beispiel

- Vertikale Fragmentierung der Relation PROJECT nach PName und Budget/Location

PROJECTS

PNo	PName	Budget	Location
P1	Database Development	200.000	Saarbr.
P2	Maintenance	150.000	Munich
P3	Web Design	100.000	Paris
P4	Customizing	250.000	Saarbr.

$$\text{PROJECTS}_1 = \pi_{\text{PNo, PName}}(\text{PROJECTS})$$
$$\text{PROJECTS}_2 = \pi_{\text{PNo, Budget, Location}}(\text{PROJECTS})$$

PROJECTS_1

PNo	PName
P1	Database Development
P2	Maintenance
P3	Web Design
P4	Customizing

PROJECTS_2

PNo	Budget	Location
P1	200.000	Saarbr.
P2	150.000	Munich
P3	100.000	Paris
P4	250.000	Saarbr.

Vertikale Fragmentierung

Heuristik zum Bestimmen der Fragmente

- Gruppieren
  - Erstelle ein Fragment für jedes Nicht-Primärschlüsselattribut
  - Vereinige zwei Fragmente gemäß einer Heuristik, bis ein bestimmtes Gütekriterium erfüllt ist
- Aufteilen
  - Beginne mit einem Fragment, das alle Attribute enthält
  - Verwende eine Heuristik zur Bestimmung von nützlichen Aufteilungen, bis ein bestimmtes Gütekriterium erfüllt ist

In der Regel liefert Aufteilen das bessere Ergebnis

Vertikale Fragmentierung

Welche Kombinationen von Attributen sollten gruppiert werden?

- Attribute, die von Anwendungen “oft zusammen verwendet” werden
- Schätze Grad der Zusammen-Verwendung mit Statistiken
  - Welche Anfragen werden gestellt?
  - Welche Attribute werden angefragt?
  - Welche Attribute werden zusammen angefragt?
  - Wie oft werden die Anfragen ausgeführt?
- Ziel
  - Bilde Cluster von Attributen, so dass verwandte Attribute (gemäß der Statistiken) im gleichen Fragment landen

Vertikale Fragmentierung

Grundlegende Techniken, um Statistiken zu sammeln

- Attribut-Verwendungs-Matrix
  Welche Anfragen verwenden welches Attribut?
- Attribut-Affinitäts-Matrix
  Wie stark stehen Attribute in Beziehung?

Attributverwendungsmatrix

- Gegeben 4 Anfragen:
  q_1 (verwendet die Attribute A_1, A_3), q_2 (A_2, A_3), q_3 (A_2, A_4), q_4 (A_3, A_4)

EXTPROJECTS

PNo	LeaderENo (A_1)	PName (A_2)	Budget (A_3)	Location (A_4)
P1	E1	Database Development	200.000	Saarbr.
P2	E1	Maintenance	150.000	Munich
P3	E3	Web Design	100.000	Paris
P4	E4	Customizing	250.000	Saarbr.

Das Primärschlüsselattribut PNo nimmt nicht am Fragmentierungsprozess teil, da es in jedes Fragment repliziert werden muss.

$$use(q_i, A_k) = \begin{cases} 1, & \text{falls } q_i \;\; A_k \text{ verwendet} \\ 0, & \text{sonst} \end{cases}$$

$$use = \begin{array}{c} \\ q_1 \\ q_2 \\ q_3 \\ q_4 \end{array} \begin{array}{c} A_1 A_2 A_3 A_4 \\ \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix} \end{array}$$

Attributverwendung und -affinität

Attribut-Affinitäts-Matrix

- Erfasse Zugriffsfrequenzen für Attributkombinationen
  Wie oft werden Attributkombinationen zugegriffen?
- Erzeuge gewichtete Attribut-Attribut-Matrix aus den Anfragestatistiken und der Attributverwendungsmatrix
- Einträge der Matrix beschreiben, wie oft ein Attribut zusammen mit einem anderen verwendet wird

Vektor der Anfragestatistiken

- Welche Anfrage wird wie oft ausgeführt?
  $qstat = (45, 5, 75, 3)$, z.B. q_2 wurde fünfmal ausgeführt
- Wir werden *qstat* manchmal als Funktion verwenden, die zu einer Anfrage ihre Statistik zurückliefert

Attribut-Affinitäts-Matrix

- Berechne Eintrag $aff(A_i, A_k)$
  - Zähle die Anfragen, die sowohl Attribut A_i als auch Attribut A_k verwenden, mit der Attributverwendungsmatrix
  - $aff(A_i, A_k) = \sum_{m \mid use(q_m, A_i)=1 \wedge use(q_m, A_k)=1} qstat(q_m)$

Beispiel

- $aff(A_1, A_3) = qstat(q_1) = 45$
- $aff(A_2, A_2) = qstat(q_2) + qstat(q_3) = 5 + 75 = 80$

$$aff = \begin{array}{c c} & \begin{array}{cccc} A_1 & A_2 & A_3 & A_4 \end{array} \\ \begin{array}{c} A_1 \\ A_2 \\ A_3 \\ A_4 \end{array} & \begin{bmatrix} 45 & 0 & 45 & 0 \\ 0 & 80 & 5 & 75 \\ 45 & 5 & 53 & 3 \\ 0 & 75 & 3 & 78 \end{bmatrix} \end{array}$$

$$use = \begin{array}{c c} & \begin{array}{cccc} A_1 & A_2 & A_3 & A_4 \end{array} \\ \begin{array}{c} q_1 \\ q_2 \\ q_3 \\ q_4 \end{array} & \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix} \end{array}$$

$$qstat = (45, 5, 75, 3)$$

Universität Trier

Idee

- Reorganisiere Spalten und Zeilen der Attribut-Affinitäts-Matrix so, dass ähnliche Einträge nah beisammen sind

Beispiel

$$
aff = \begin{array}{c} \\ \mathrm{A}_1 \\ \mathrm{A}_3 \\ \mathrm{A}_2 \\ \mathrm{A}_4 \end{array}
\begin{array}{c} \begin{array}{cccc} A_1 & A_3 & A_2 & A_4 \end{array} \\ \begin{bmatrix} 45 & 45 & 0 & 0 \\ 45 & 53 & 5 & 3 \\ 0 & 5 & 80 & 75 \\ 0 & 3 & 75 & 78 \end{bmatrix} \end{array}
$$

Resultierende Fragmentierung für die geclusterte Affinitätsmatrix der vorherigen Folie

EXTPROJECTS

PNo	LeaderENo (A_1)	PName (A_2)	Budget (A_3)	Location (A_4)
P1	E1	Database Development	200.000	Saarbr.
P2	E1	Maintenance	150.000	Munich
P3	E3	Web Design	100.000	Paris
P4	E4	Customizing	250.000	Saarbr.

EXTPROJECTS_1

PNo	LeaderENo (A_1)	Budget (A_3)
P1	E1	200.000
P2	E1	250.000
P3	E3	100.000
P4	E4	250.000

EXTPROJECTS_2

PNo	PName (A_2)	Location (A_4)
P1	Database Development	Saarbr.
P2	Maintenance	Munich
P3	Web Design	Paris
P4	Customizing	Saarbr.

Clusteringalgorithmus [Hoffer and Severance, 1975]

- Globales Affinitätsmaß *AM* als Optimierungsziel

$$AM = \sum_{i=1}^{n} \sum_{k=1}^{n} aff(A_i, A_k)[aff(A_i, A_{k-1}) + aff(A_i, A_{k+1}) + aff(A_{i-1}, A_k) + aff(A_{i+1}, A_k)]$$

wobei

$$aff(A_0, A_k) = aff(A_i, A_0) = aff(A_{n+1}, A_k) = aff(A_i, A_{n+1}) = 0$$

Die unteren Bedingungen kümmern sich um die Fälle, in denen ein Attribut links vom linkesten oder rechts vom rechtesten Attribut platziert werden soll.

wegen der Symmetrie:

$$AM = \sum_{i=1}^{n} \sum_{k=1}^{n} aff(A_i, A_k)[aff(A_i, A_{k-1}) + aff(A_i, A_{k+1})]$$

Wir lassen hier einen Faktor 2 weg, der für die Optimierung keine Rolle spielt.

Definition 3.2 (Bindung)

Bindung zwischen den Attributen A_x und A_y

$$bond(A_x, A_y) = \sum_{z=1}^{n} aff(A_z, A_x) \cdot aff(A_z, A_y)$$

Dies entspricht dem Skalarprodukt der Spalten der Attributaffinitätsmatrix, die A_x und A_y entsprechen.

Reformulierung von AM mit bonds:

$$AM = \sum_{j=1}^{n} [bond(A_j, A_{j-1}) + bond(A_j, A_{j+1})]$$

Clusteringalgorithmus

Schritte

- Initialisierung
  - Platziere die ersten beiden Spalten der Attribut-Affinitäts-Matrix (*AM*) in die ersten beiden Spalten der geclusterten Affinitätsmatrix (*CA*)
- Iteration
  - Wähle eine der verbleibenden Spalten
  - Versuche sie in eine der übrigen Positionen in *CA* zu platzieren
  - Wähle die Platzierung, die den größten *Beitrag* zum globalen Affinitätsmaß liefert
- Zeilenanordnung
  - Vertausche die Zeilen, so dass ihre relativen Positionen den relativen Positionen der Spalten entsprechen

Beitrag zum globalen Affinitätsmaß, wenn Attribut A_m zwischen A_i und A_k platziert wird:

$$\begin{aligned} cont(A_i, A_m, A_k) &= AM_{new} - AM_{old} \\ &= 2 \cdot bond(A_i, A_m) + 2 \cdot bond(A_m, A_k) - 2 \cdot bond(A_i, A_k) \end{aligned}$$

$2 \cdot bond(A_i, A_k)$ gehen verloren, weil A_i seinen alten rechten Nachbarn verliert und A_k seinen alten linken Nachbarn verliert; die Argumentation für die beiden übrigen Summanden ist ähnlich.

Beim Einfügen von A_m am linken oder rechten Rand wächst das Affinitätsmaß immer um $2 \cdot bond(A_m, A)$, wobei A das Attribut am Rand ist.

Beispiel

$$aff = \begin{array}{c} \\ A_1 \\ A_2 \\ A_3 \\ A_4 \end{array} \begin{array}{c} \begin{array}{cccc} A_1 & A_2 & A_3 & A_4 \end{array} \\ \begin{bmatrix} 45 & 0 & 45 & 0 \\ 0 & 80 & 5 & 75 \\ 45 & 5 & 53 & 3 \\ 0 & 75 & 3 & 78 \end{bmatrix} \end{array} \qquad CA = \begin{array}{c} \\ A_1 \\ A_2 \\ A_3 \\ A_4 \end{array} \begin{array}{c} \begin{array}{cccc} A_1 & A_2 & - & - \end{array} \\ \begin{bmatrix} 45 & 0 & - & - \\ 0 & 80 & - & - \\ 45 & 5 & - & - \\ 0 & 75 & - & - \end{bmatrix} \end{array}$$

Beitrag, wenn A_4 zwischen A_1 und A_2 eingefügt wird

$$cont(A_i, A_m, A_k) = 2 \cdot bond(A_i, A_m) + 2 \cdot bond(A_m, A_k) - 2 \cdot bond(A_i, A_k)$$

$$cont(A_1, A_4, A_2) = 2 \cdot bond(A_1, A_4) + 2 \cdot bond(A_4, A_2) - 2 \cdot bond(A_1, A_2)$$

$$bond(A_x, A_y) = \sum_{z=1}^{n} aff(A_z, A_x) \cdot aff(A_z, A_y)$$

$$\begin{aligned} bond(A_1, A_4) = {} & aff(A_1, A_1) \cdot aff(A_1, A_4) + \\ & aff(A_2, A_1) \cdot aff(A_2, A_4) + \\ & aff(A_3, A_1) \cdot aff(A_3, A_4) + \\ & aff(A_4, A_1) \cdot aff(A_4, A_4) \end{aligned}$$

Universität Trier

Beispiel

$$aff = \begin{array}{c} \\ \mathrm{A}_1 \\ \mathrm{A}_2 \\ \mathrm{A}_3 \\ \mathrm{A}_4 \end{array} \begin{array}{c} \begin{array}{cccc} A_1 & A_2 & A_3 & A_4 \end{array} \\ \begin{bmatrix} 45 & 0 & 45 & 0 \\ 0 & 80 & 5 & 75 \\ 45 & 5 & 53 & 3 \\ 0 & 75 & 3 & 78 \end{bmatrix} \end{array} \qquad CA = \begin{array}{c} \\ \mathrm{A}_1 \\ \mathrm{A}_2 \\ \mathrm{A}_3 \\ \mathrm{A}_4 \end{array} \begin{array}{c} \begin{array}{cccc} A_1 & A_2 & - & - \end{array} \\ \begin{bmatrix} 45 & 0 & - & - \\ 0 & 80 & - & - \\ 45 & 5 & - & - \\ 0 & 75 & - & - \end{bmatrix} \end{array}$$

$$cont(A_1, A_4, A_2) = 2 \cdot 135 + 2 \cdot 11865 - 2 \cdot 225 = 23550$$

$$cont(-, A_4, A_1) = 2 \cdot 135 = 270$$

$$cont(A_2, A_4, -) = 2 \cdot 11865 = 23730$$

Wir fügen A_4 also am rechten Rand von CA ein.

Beispiel

$$
aff = \begin{array}{c} \\ \mathrm{A}_1 \\ \mathrm{A}_2 \\ \mathrm{A}_3 \\ \mathrm{A}_4 \end{array}
\begin{array}{c} \begin{array}{cccc} A_1 & A_2 & A_3 & A_4 \end{array} \\ \begin{bmatrix} 45 & 0 & 45 & 0 \\ 0 & 80 & 5 & 75 \\ 45 & 5 & 53 & 3 \\ 0 & 75 & 3 & 78 \end{bmatrix} \end{array}
\qquad
CA = \begin{array}{c} \\ \mathrm{A}_1 \\ \mathrm{A}_2 \\ \mathrm{A}_3 \\ \mathrm{A}_4 \end{array}
\begin{array}{c} \begin{array}{cccc} A_1 & A_2 & A_4 & - \end{array} \\ \begin{bmatrix} 45 & 0 & 0 & - \\ 0 & 80 & 75 & - \\ 45 & 5 & 3 & - \\ 0 & 75 & 78 & - \end{bmatrix} \end{array}
$$

$$cont(A_1, A_3, A_2) = 2 \cdot 4410 + 2 \cdot 890 - 2 \cdot 225 = 10250$$

$$cont(A_2, A_3, A_4) = 2 \cdot 890 + 2 \cdot 768 - 2 \cdot 11865 = -20214$$

$$cont(-, A_3, A_1) = 2 \cdot 4410 = 8820$$

$$cont(A_4, A_3, -) = 2 \cdot 768 = 1536$$

Wir fügen A_3 also zwischen A_1 und A_2 ein.

Umordnen der Zeilen ergibt genau die Matrix von Folie 168.

Ableiten einer Fragmentierung aus der geclusterten Affinitätsmatrix

$$\begin{array}{c}
 \\ \mathrm{A}_1 \\ \mathrm{A}_3 \\ \mathrm{A}_2 \\ \mathrm{A}_4
\end{array}
\begin{array}{c}
\begin{array}{cccc} A_1 & A_3 & A_2 & A_4 \end{array} \\
\begin{bmatrix} 45 & 45 & 0 & 0 \\ 45 & 53 & 5 & 3 \\ 0 & 5 & 80 & 75 \\ 0 & 3 & 75 & 78 \end{bmatrix}
\end{array}$$

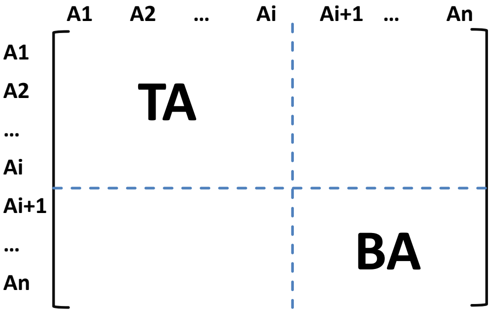
Erkennen guter Cluster ist nichttrivial für einen Algorithmus

- Wie viele Cluster?
- Welche Clustergrenzen?

Wir diskutieren nun einen Algorithmus, um zwei Cluster zu finden. Mehr Cluster kann man (zum Beispiel) finden, indem man den Algorithmus rekursiv anwendet.

Ziel

- Bestimmen eines Attributs A_i, bei dem *CA* aufgeteilt wird in *TA* (mit den Attributen A_1 bis A_i) und *BA* (mit den Attributen A_{i+1} bis A_n)
- Beide Submatrizen werden zu einem Fragment
- Auswahl basierend auf Minimierung von Inter-Fragment-Zugriffen von Anfragen



Notation

- Q Menge der Anfragen
- $AQ(q_i) := \{A_k | use(q_i, A_k) = 1\}$ Menge der Attribute, die q_i verwendet
- $TQ := \{q_i | AQ(q_i) \subseteq TA\}$ Anfragen, die nur Attribute aus TA verwenden
- $BQ := \{q_i | AQ(q_i) \subseteq BA\}$ Anfragen, die nur Attribute aus BA verwenden
- $OQ := Q \setminus (TQ \cup BQ)$ übrige Anfragen

Definiere verschiedene Kostenmaße

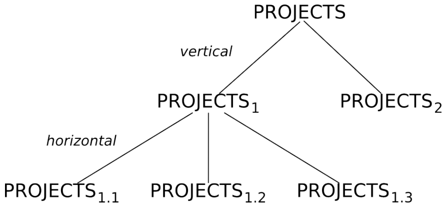
- $CTQ := \sum_{q_i \in TQ} qstat(q_i)$ Kosten von Anfragen, die nur Attribute aus TA verwenden
- $CBQ := \sum_{q_i \in BQ} qstat(q_i)$ Kosten von Anfragen, die nur Attribute aus BA verwenden
- $COQ := \sum_{q_i \in OQ} qstat(q_i)$ Kosten von Anfragen, die Attribute aus beiden verwenden

Bestimme A_i so dass $CTQ * CBQ - COQ^2$ maximiert wird

Hybride Fragmentierung

Hybride Fragmentierung

- Ein Fragment einer Relation ist selbst wieder eine Relation
- Fragmente können wieder fragmentiert werden
- Kombination aus horizontaler und vertikaler Fragmentierung möglich



$$\begin{aligned}
\text{PROJECTS}_1 &= \pi_{\text{PNo, PName, Location}}(\text{PROJECTS}) \\
\text{PROJECTS}_2 &= \pi_{\text{PNo, Budget}}(\text{PROJECTS}) \\
\text{PROJECTS}_{1.1} &= \sigma_{\text{Location}=\text{'Saarbr.'}}(\text{PROJECTS}_1) \\
\text{PROJECTS}_{2.1} &= \sigma_{\text{Location}=\text{'Munich'}}(\text{PROJECTS}_1) \\
\text{PROJECTS}_{3.1} &= \sigma_{\text{Location}=\text{'Paris'}}(\text{PROJECTS}_1)
\end{aligned}$$

$$\begin{aligned}
\text{PROJECTS} = \\
&(\text{PROJECTS}_{1.1} \cup \text{PROJECTS}_{1.2} \cup \text{PROJECTS}_{1.3}) \bowtie \text{PROJECTS}_2
\end{aligned}$$

Zusammenfassung der Fragmentierungsvarianten

- Primäre horizontale Fragmentierung
  Horizontale Fragmentierung der Relation R wird definiert mit Prädikaten, die sich auf Relation R beziehen
- Abgeleitete horizontale Fragmentierung
  Horizontale Fragmentierung der Relation R wird definiert mit Prädikaten, die sich auf eine andere Relation S beziehen
- Vertikale Fragmentierung
  Fragmentierung der Attribute von Relation R
- Hybride Ansätze
  Kombination verschiedener Fragmentierungsmethoden

Allokation und Replikation

Grundlagen

- Nachdem die Fragmente bestimmt sind, müssen sie Rechnern im Netz zugewiesen werden
  - Unterschiedliche Allokationsstrategien
  - Leistungsgewinn vs. Replikation
- Effizienz
  - Minimierung der Kosten durch entferntes Lesen/Schreiben
  - Vermeiden von Engpässen
- Verlässlichkeit
  - Auswahl von Rechnern auf Basis ihrer Verfügbarkeit
  - Redundantes Speichern von Daten

Allokation

- *Nicht-redundant*
  Kein Fragment wird mehr als einmal alloziert $\rightarrow$ *partitionierte Datenbank*
- *Redundant*
  (Einige) Fragmente werden mehr als einmal alloziert $\rightarrow$ *Replizierte Datenbank*

Replikation

- *Volle Replikation*
  Jede globale Relation wird auf jedem Rechner gespeichert
  - Keine Strategie zur Fragmentierung und Allokation notwendig
  - Keine verteilte Anfrageausführung
  - Hoher Speicherplatzbedarf
  - Hohe Änderungskosten
- *Teilweise Replikation*
  sonst

Goldene Regeln der Allokation

- Platziere Daten möglichst nah an der Stelle, wo sie benutzt werden
- Verwende Lastbalancierung, um die globale Systemperformance zu optimieren

Strategien

- Optimale Allokation [Dadam, 1996]
  - Nicht-redundant
  - Redundant
- Standardverfahren [Toerey, 1999]
  - Nicht-redundante "best fit" Methode
  - "Alle gewinnbringenden (beneficial) Rechner" Methode
  - Progressive Allokationsmethode

Optimale Allokation

Nicht-redundante optimale Allokation

Ziel

- Minimiere Speicherkosten ($\sum_S$) und Transferkosten ($\sum_T$) für P Fragmente und K Rechner

Kostenmodell für **nicht-redundante Allokation**

- Speicherkosten

$$\sum\nolimits_S = \sum_{p,i} G_p V_{pi} S_i$$

  - G_p: Größe von Fragment P in Dateneinheiten
  - S_i: Speicherkosten pro Dateneinheit an Rechner i
  - V_{pi}: Allokation der Fragmente;
    $V_{pi} = 1$ wenn Fragment p an Knoten i alloziert wurde, 0 sonst

- Transferkosten

$$\sum_T = \underbrace{\sum_{i,t,p,k} H_{it} O_{tp} V_{pk} T_{ik}}_{\text{transferiere Op. von } i \text{ zu } k} + \underbrace{\sum_{i,t,p,k} H_{it} R_{tp} V_{pk} T_{ki}}_{\text{Transferiere Erg. von } k \text{ zu } i}$$

  - H_{it}: Frequenz von Operationen des Typs t auf Rechner i
  - O_{tp}: Größe einer Operation vom Typ t auf Fragment p in Dateneinheiten (Länge des Anfragestrings)
  - T_{ik}: Transferkosten in Dateneinheit von Rechner i auf Rechner k
  - R_{tp}: Größe des Ergebnisses einer Operation vom Typ t auf Fragment p

Nichtredundante optimale Allokaion

Randbedingungen

$$\sum_i V_{pi} = 1 \;\text{ für }\; p = 1, \ldots, P$$

$$\sum_p G_p V_{pi} \leq M_i \;\text{ für }\; i = 1, \ldots, K$$

wobei M_i die maximale Speicherkapazität von Rechner i in Dateneinheiten angibt

Optimierungsproblem

- Bestimme das Minimum von $\sum_S + \sum_T$ unter den obigen Randbedingungen
- Oft werden Heuristiken angewendet
  - Bestimmen von relevanten Kandidatenlösungen
  - Bestimmen der Kosten für diese Kandidaten
- Da das Problem ein ganzzahliges lineares Programm ist (ILP), gibt es effiziente approximative Lösungsverfahren

Redundante optimale Allokation

Kostenmodell für *redundante Allokation* [Dadam, 1996] erfordert andere Randbedingungen

$$\sum_i V_{pi} \geq 1 \text{ für } p = 1, \ldots, P$$

$$\sum_p G_p V_{pi} \leq M_i \text{ für } i = 1, \ldots, K$$

. . . und andere Transferkosten

$$\sum\nolimits_T = \sum_{i,t,p} H_{it} \underset{k:V_{pk}=1}{\Phi_t} (O_{tp} T_{ik} + R_{tp} T_{ki})$$

- Leseoperation (die Fragment p liest) wird auf dem Rechner i mit geringsten Kosten ausgeführt
- Änderungsoperation (die Fragment p ändert) wird auf allen Rechnern ausgeführt, wo Fragment p alloziert wurde
- Φ_t: repräsentiert $\sum$ für Änderungsoperationen oder $\min$ für Leseoperationen

Allokationsalgorithmen

Nichtredundante Best-Fit-Methode

Grundlegende Frage:
Was ist der beste Rechner für jedes Fragment?

- Benefit: Gesamtzahl der lokalen Anfragen und Änderungen an ein Fragment
- Fragment R_i wird auf Rechner S_k alloziert
  S_k ist der Rechner mit der größten Zahl von lokalen Anfragen und Änderungen gegen Fragment R_i
- Algorithmus
  - Gruppiere Zugriffe auf ein Fragment (lesend/ändernd) nach Rechnern
  - Wähle den Rechner zur Allokation mit der größten Zugriffszahl
  - Wenn es mehrere äquivalente Optionen gibt, wähle den Rechner mit der kleinsten Zahl von Fragmenten

Wir sprechen auch von der Allokation eines Fragmentes auf den Rechner mit den meisten Zugriffen.

Nichtredundante Best-Fit-Methode

Beispiel

Fragment	Rechner	# Zugriffe (r/w)
R_1	S_1	12
	S_2	2
R_2	S_3	27
R_3	S_1	12
	S_2	12

Resultierende Allokation

- Alloziere Fragment R_1 auf Rechner S_1
- Alloziere Fragment R_2 auf Rechner S_3
- Alloziere Fragment R_3 auf Rechner S_2 um die Robustheit zu erhöhen

Universität Trier

Nichtredundante Best-Fit-Methode

Vor- und Nachteile

- Geringer Rechenaufwand
- Recht ungenau, da die reine Zugriffszahl nicht I/O-Aufwand (lesen/schreiben), Größe der Daten etc. berücksichtigt
- berücksichtigt keine Replikation

Alle gewinnbringenden (beneficial) Rechner

Aspekte

- Verwendet Redundanz bzw. Replikation
- betrachtet echte Kosten für Lesen/Schreiben und Netzzugriff

Wesentliche Idee

- Vergleiche den **Gewinn** durch eine zusätzliche Kopie mit den **Kosten** um diese Kopie aktuell zu halten
- Alloziere auf allen Rechnern, wo der Gewinn größer als die Kosten für eine zusätzliche Kopie ist

Alle gewinnbringenden (beneficial) Rechner

Gewinn aus einer Kopie von Fragment R_i auf Rechner S_k

- Differenz der Ausführungszeit einer **Anfrage auf einem anderen Rechner und einer lokalen Anfrage** an Rechner S_k
- multipliziert mit der Frequenz von Anfragen an Fragment R_i, die von Rechner S_k ausgehen

Kosten einer zusätzlichen Kopie von Fragment R_i auf Rechner S_k

- Die Ausführungszeit für **alle lokalen Änderungen** an Fragment R_i, die von Rechner S_k ausgehen
- plus die Ausführungszeit für **alle Änderungen auf einem anderen Rechner** an Fragment R_i

Wesentliche Schritte

- **Berechne Kosten und Gewinn** für alle Fragmente und alle Rechner
- Alloziere ein Fragment auf allen Rechnern, **wo der Gewinn höher als die Kosten ist**

Das Verfahren kann auch für nicht-redundante Allokation verwendet werden

Kosten

Fragment	Rechner	Änderungen von Rechnern	Kosten	Gesamtkosten
R_1	S_1	$2^* S_1$, S_2	$2 \times 150ms + 1 \times 600ms$	$900ms$
	S_2	$2^* S_1$, S_2	$2 \times 600ms + 1 \times 150ms$	$1350ms$
	S_3	$2 * S_1, S_2$	$3 \times 600ms$	$1800ms$
R_2	S_1	S_1	$1 \times 200ms$	$200ms$
	S_2	S_1	$1 \times 700ms$	$700ms$
	S_3	S_1	$1 \times 700ms$	$700ms$
R_3	S_1	S_2, S_3	$2 \times 1100ms$	$2200ms$
	S_2	S_2, S_3	$1 \times 250ms + 1 \times 1100ms$	$1350ms$
	S_3	S_2, S_3	$1 \times 1100ms + 1 \times 250ms$	$1350ms$

Kosten: Ausführungszeiten für lokale und entfernte Änderungsoperationen

Alle gewinnbringenden (beneficial) Rechner – Beispiel

Gewinn

Fragment	Rechner	Gewinn	Gesamtgewinn
R_1	S_1	$2 \times (500ms - 100ms)$	$800ms$
	S_2	$1 \times (500ms - 100ms)$	$400ms$
	S_3	$0 \times (500ms - 100ms)$	$0ms$
R_2	S_1	$1 \times (650ms - 150ms)$	$500ms$
	S_2	$0 \times (650ms - 150ms)$	$0ms$
	S_3	$3 \times (650ms - 150ms)$	$1500ms$
R_3	S_1	$0 \times (1000ms - 200ms)$	$0ms$
	S_2	$2 \times (1000ms - 200ms)$	$1600ms$
	S_3	$3 \times (1000ms - 200ms)$	$2400ms$

Gewinn: Differenz der Ausführungszeit von lokalen und entfernten Leseoperationen

Alloziere Fragmente auf Rechnern, für die Gewinn $>$ Kosten; jedes Fragment aber mindestens einmal.

Fragment	Rechner	Gewinn	Kosten
R_1	S_1	$800ms$	$900ms$
	S_2	$400ms$	$1350ms$
	S_3	$0ms$	$1800ms$
R_2	S_1	$500ms$	$200ms$
	S_2	$0ms$	$700ms$
	S_3	$1500ms$	$700ms$
R_3	S_1	$0ms$	$2200ms$
	S_2	$1600ms$	$1350ms$
	S_3	$2400ms$	$1350ms$

Alle gewinnbringenden (beneficial) Rechner

Vor- und Nachteile

- Einfacher Algorithmus
- Globale Durchschnitte von Anfrage- und Änderungszeiten oft unrealistisch
- Netztopologie und -Protokolle unberücksichtigt

Progressive (schrittweise) Allokation

Idee

- Erweiterung der “alle gewinnbringenden Rechner”-Methode
- Berücksichtigt, dass sich Allokationsentscheidungen beeinflussen können, z.B. durch Verzögerungen in der Netzlaufzeit

Prinzip

- Erste Kopie eines Fragments wird immer auf dem Rechner alloziert, der Gewinn minus Kosten maximiert (wie in der “alle gewinnbringenden Rechner”-Methode)
- Die folgende Allokationsentscheidung berücksichtigt den Ort der ersten Kopie und maximiert wieder die Differenz von Gewinn und Kosten für die übrigen Rechner
- Terminiert, wenn Gewinn für alle Rechner nicht größer als Kosten ist

Universität Trier

Beispiel

Fragment	Rechner	Gewinn	Kosten	(Gewinn - Kosten)
R_1	S_1	$800ms$	$900ms$	$-100ms$
	S_2	$400ms$	$1350ms$	$-950ms$
	S_3	$0ms$	$1800ms$	$-1800ms$
R_2	S_1	$500ms$ $400ms$	$200ms$	$300ms$ $200ms$
	S_2	$0ms$	$700ms$	$-700ms$
	S_3	$1500ms$	$700ms$	$800ms$
R_3	S_1	$0ms$	$2200ms$	$-2200ms$
	S_2	$1600ms$ $1200ms$	$1350ms$	$250ms$ $-150ms$
	S_3	$2400ms$	$1350ms$	$1050ms$

Progressive (schrittweise) Allokation

Vor- und Nachteile

- Progressive Optimierung anstelle von unabhängigen Entscheidungen
- Kosten bleiben unverändert bei jeder Entscheidung
  weil die Änderung eines zusätzlichen Fragments unabhängig von den bisher allozierten Fragmenten ist
- Gewinn bleibt nicht konstant
  Gewinn wird kleiner, wenn eine neue Kopie näher an einem gegebenen Rechner alloziert wird (kleinere Übertragungsverzögerung)
- Berücksichtigt nicht die Balancierung von Last bzw. Speichergröße zwischen den Rechnern

Graphbasierte Fragmentierung und Allokation

Übersicht des Schism-Ansatzes [Curino, 2010]

1. Sammle (viel) Workloadinformation in Form von gelesenen und geschriebenen Tupeln pro Transaktion
2. Erstelle einen Graphen, der die Datenbank und den Workload repräsentiert
3. Partitionere den Graphen in so viele Partitionen, wie es Rechner gibt
4. Finde Erklärung der Partitionierung durch einfache(re) Regeln
5. Prüfe, ob einfachere Methode (volle Replikation oder Hash-Partitionierung) besser funktioniert
6. Implementiere die erhaltene Partitionierung

Zusätzliche Optimierungsmöglichkeit: Replikation

Darstellung von Daten und Workload als Graph

Definition des Graphen

- **Knoten**: Zu jedem Tupel in der Datenbank gibt es einen Knoten im Graphen
- **Kanten**: Es gibt eine Kante zwischen zwei Tupeln, wenn sie in der gleichen Transaktion zugegriffen werden
- **Kantengewichte**: Das Gewicht einer Kante entspricht der Anzahl von Transaktionen, in der die beiden Tupel zugegriffen werden

Partitionierung des Graphen

- Partitioniere den Graphen in disjunkte Teilmengen von Knoten, so dass
  - das aggregierte Gewicht von Kanten zwischen Partitionen minimiert wird
  - die Partitionen (fast) die gleiche Größe haben

Für dieses Problem gibt es viele Standardverfahren

http://glaros.dtc.umn.edu/gkhome/views/metis

GRAPH REPRESENTATION

transaction edges

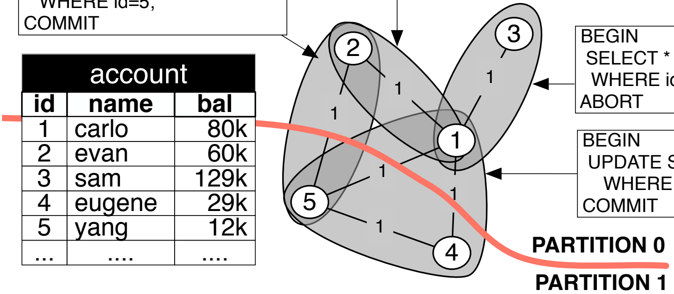
```
BEGIN
 UPDATE account SET bal=60k
   WHERE id=2;
 SELECT * FROM account
   WHERE id=5;
COMMIT
```

```
BEGIN
UPDATE account SET bal=bal-1k WHERE name="carlo";
UPDATE account SET bal=bal+1k WHERE name="evan";
COMMIT
```

```
BEGIN
 SELECT * FROM account
  WHERE id IN {1,3}
ABORT
```

```
BEGIN
 UPDATE SET bal=bal+1k
    WHERE bal < 100k;
COMMIT
```

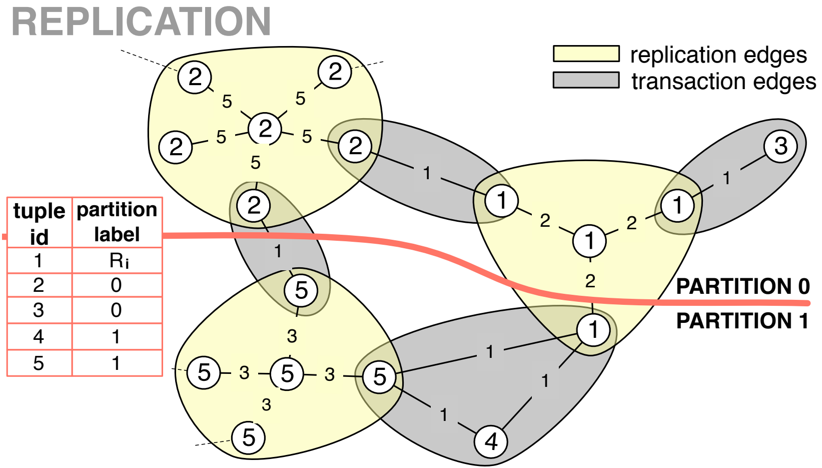
account		
id	**name**	**bal**
1	carlo	80k
2	evan	60k
3	sam	129k
4	eugene	29k
5	yang	12k
...	....	....



Quelle [Curino, 2010]

Unterstützung von Replikation im Graphen

- Expandiere Knoten jedes Tuples, das von mindestens einer Transaktion geändert wird
- Expandiere Knoten in einen Zentralknoten und *n* “Brückenknoten”, die die *n* Transaktionen repräsentieren, in denen dieses Tupel zugegriffen wird
- Das Gewicht der inneren Kanten entspricht der Anzahl der Transaktionen, in denen dieses Tupel geändert wird
- Wenn alle Knoten eines Tupels der gleichen Partition zugewiesen werden, wird das Tupel nicht repliziert; andernfalls wird es in allen Partitionen repliziert, wo mindestens einer der Knoten gelandet ist
- Wenn eine der eingeführten inneren Kanten durch die Partitionierung zerschnitten wird, entspricht ihr Gewicht den zusätzlichen Transferkosten für die Übertragung der Änderungen dieses Tupels



tuple id	partition label
1	R_i
2	0
3	0
4	1
5	1

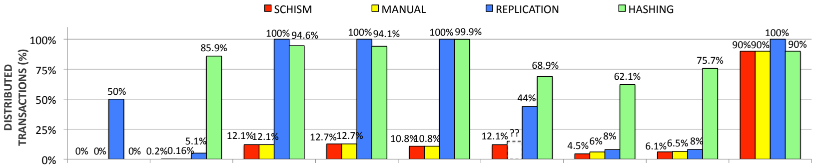
Quelle [Curino, 2010]

Erklären der Partitionierung

- Die Zuordnungstabelle von Tupeln zu Partitionen ist oft zu groß zum Speichern
- Lösung: Lernen einer Funktion, die Primärschlüsselwerte auf Partitionen abbildet
- Einfacher Ansatz: Klassifikator auf Basis eines Entscheidungsbaums

Beispiel (von der vorherigen Folie)

$$\begin{aligned}(id = 1) &\mapsto partitions = \{0, 1\}\\(2 \leq id < 4) &\mapsto partitions = \{0\}\\(id \geq 4) &\mapsto partitions = \{1\}\end{aligned}$$



dataset:	YCSB-A	YCSB-E	TPCC-2W	TPCC-2W	TPCC-50W	TPC-E	EPINIONS	EPINIONS	RANDOM
partitions:	any	100	2	2	10	2	2	10	10
coverage:	any	1%	50%	0.5%	1%	5%	15%	15%	any
SCHISM:	hashing	range-predicates	range-predicates	range-predicates	range-predicates	range-predicates	look-up table	look-up table	hashing

Quelle [Curino, 2010]

Zusammenfassung

Zusammenfassung

- Fragmentierung
  - Horizontal (primär, abgeleitet), Algorithmen
  - Vertikal
  - Hybrid
- Allokation und Replikation
  - Optimaler Allokationsalgorithmus
  - Nichtredundante Best-Fit-Methode
  - Alle gewinnbringenden Rechner
  - Progressive Allokation
- Graphbasierte Fragmentierung und Allokation

Referenzen

[Özsu Valduriez, 2011] M. Tamer Özsu, P. Valduriez.
Principles of Distributed Database Systems.
Third Edition, Springer, 2011.

[Hoffer and Severance, 1975] J. Hoffer and D. Severance.
The use of cluster analysis in physical data base design.
VLDB 1975, S. 69–86.

[Dadam, 1996] P. Dadam.
Verteilte Datenbanken und Client/Server-Systeme.
Springer-Verlag, Berlin, Heidelberg 1996.

[Toerey, 1999] Toby J. Teorey
Database modeling and design
Third Edition, Morgan Kaufmann Publishers, San Francisco, CA, 1999.

[Curino, 2010] C. Curino, E. Jones, Y. Zhang, and S. Madden.
Schism: a Workload-Driven Approach to Database Replication and Partitioning.
VLDB 2010, S. 48–57.