Motivation

Aufgabe der Anfrageverarbeitung ist es . . .

. . . Anfragen zu verarbeiten

Beispiel

- Wie viele Studenten gibt es an der Universität Trier?
- Antwort: 13.331

Weitere Anforderungen

- Niedrige Antwortzeiten
- Hoher Durchsatz von Anfragen
- Effiziente Verwendung von Hardware
- . . .

Motivation

Unterschiede zur zentralen Anfrageverarbeitung

- Berücksichtigung der physischen Datenverteilung während der Anfrageoptimierung
- Berücksichtigung von Kommunikationskosten

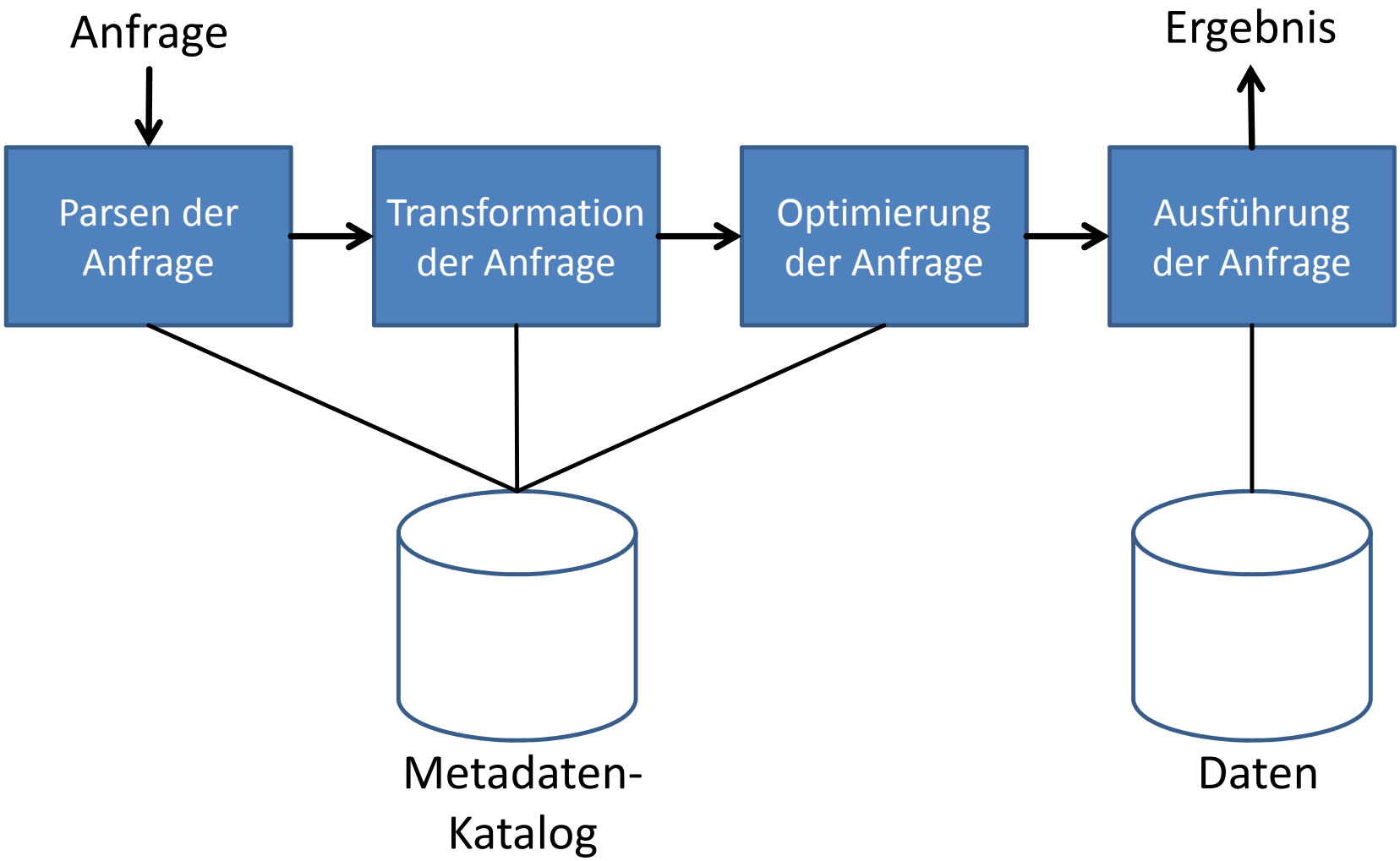
Annahmen

- Daten sind über mehrere Rechner verteilt
- Alle Rechner verwenden das gleiche globale konzeptuelle Schema
- Anfragen werden gegen das globale Schema gestellt

Wiederholung: Zentrale Anfrageverarbeitung

Phasen zentraler Anfrageverarbeitung

Workflow der zentralen Anfrageverarbeitung

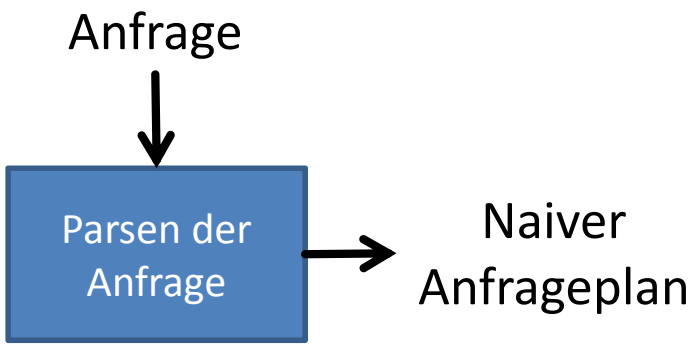


Parsen der Anfrage

Parsen der Anfrage

Übersetzen einer deklarativen Anfrage in eine interne Repräsentation

- Anfrage in einer deklarativen Sprache formuliert, z.B. SQL
- Der Parser übersetzt die Anfrage in eine interne Repräsentation
  - Das Ergebnis heißt auch naiver Anfrageplan
  - Der Plan wird durch einen Baum von Operatoren der relationalen Algebra beschrieben



Beispiel

- Datenbank mit Informationen über Angestellte und Projekte
  - EMPLOYEES(*EID*, *EName*, *Title*)
  - ASSIGNMENT(*ENo*, *PNo*, *Duration*)
- Anfrage: Bestimme die Namen aller Angestellten, die für Projekt ’P1’ arbeiten
  - **SELECT** EName
    FROM Employees e, Assignment a
    WHERE e.EID = ENo **AND** PNo=’P1’

Beispiel

Anfrage

- **SELECT** EName
  FROM Employees e, Assignment a
  WHERE e.EID = ENo **AND** PNo='P1'

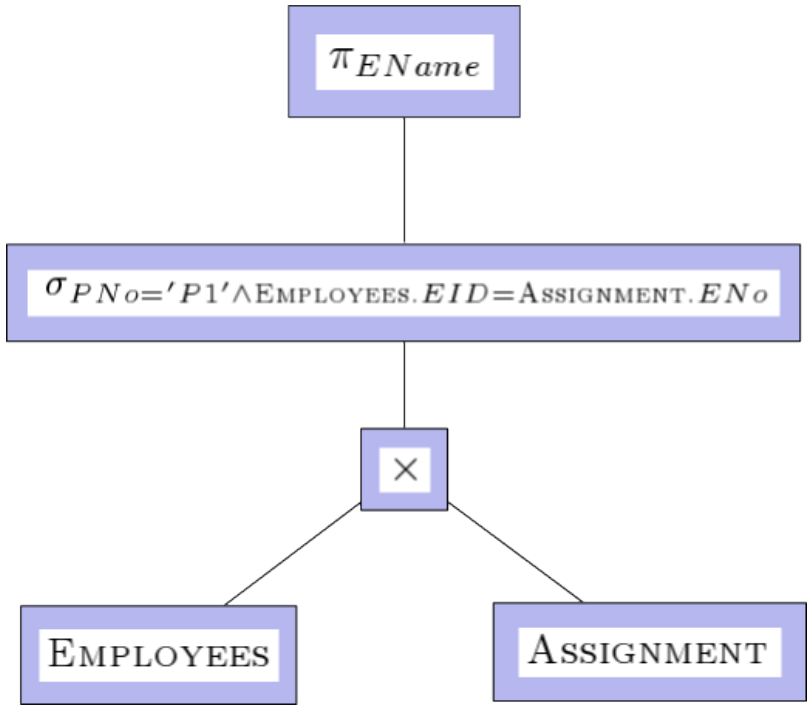
Übersetzung in relationale Algebra

$$\pi_{EName}(\sigma_{PNo='P1' \wedge \textsc{Employees}.EID=\textsc{Assignment}.ENo}(\textsc{Employees} \times \textsc{Assignment}))$$

Im Gegensatz zur SQL-Anfrage enthält der Algebraausdruck bereits die notwendigen grundlegenden Operatoren zur Ausführung der Anfrage

Operatorbaum

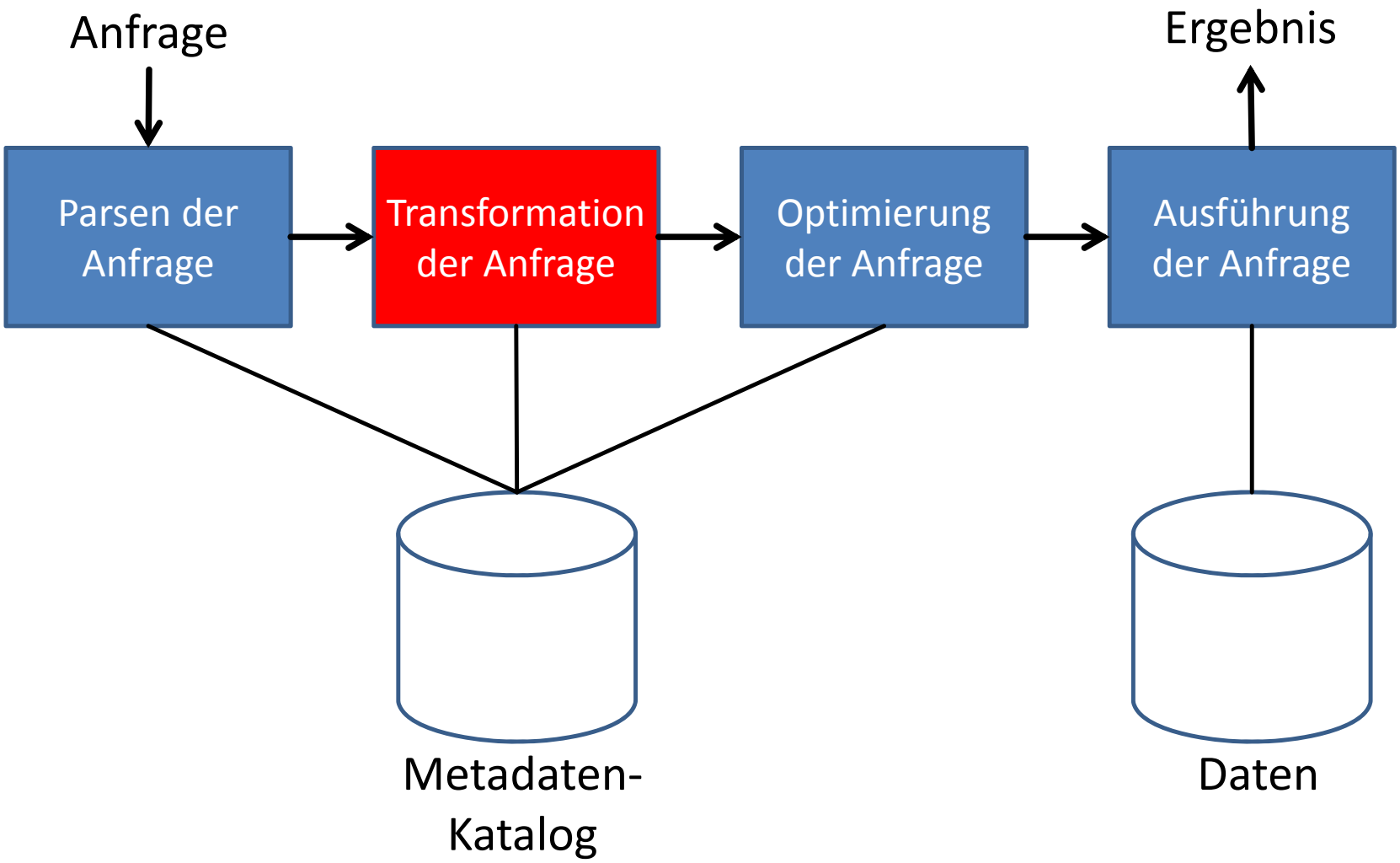
$$\pi_{EName}(\ \sigma_{PNo='P1' \wedge \text{EMPLOYEES}.EID=\text{ASSIGNMENT}.ENo}\ (\text{EMPLOYEES} \times \text{ASSIGNMENT}))$$



Operatorbaum

Transformation der Anfrage

Workflow der zentralen Anfrageverarbeitung



Transformation der Anfrage

Schritte

1. Namensauflösung
   Umwandeln von Objektnamen in interne Namen
2. Semantische Analyse
   Prüfen der globalen Relationen und Attribute, Expansion von Views, globale Zugriffskontrolle
3. Normalisierung
   Umwandeln von Prädikaten in ein kanonisches Format
4. Einfache algebraische Umformungen
   Anwendung von Heuristiken, um schlechte Pläne zu eliminieren

Semantische Analyse

- Prüfe, ob alle Attribute und Relationen, die in der Anfrage verwendet werden, im **globalen Schema** definiert sind
- Wenn die Anfrage auf einer **View** definiert ist, **ersetze Referenzen** von Relationen/Attributen durch Relationen/Attribute im globalen Schema
- Führe **einfache Integritätschecks** durch, z.B. ob Attribute in Vergleichen mit dem korrekten Typ verwendet werden
- Erster Test, ob die Anfrage (bzw. der anfragende Benutzer) die **Rechte** hat, um die gewünschten Relationen/Attribute zuzugreifen

Normalisierung

Ziel

- Vereinfachung der folgenden Optimierungen durch Umformen der Anfrage in ein **kanonisches Format**
- Selektions- und Joinprädikate
  - Konjunktive Normalform vs. disjunktive Normalform
  - Konjunktive Normalform:
    $(p_{11} \vee p_{12} \vee \cdots \vee p_{1n}) \wedge \cdots \wedge (p_{m1} \vee p_{m2} \vee \cdots \vee p_{mn})$
  - Disjunktive Normalform: $(p_{11} \wedge p_{12} \wedge \cdots \wedge p_{1n}) \vee \cdots \vee (p_{m1} \wedge p_{m2} \wedge \cdots \wedge p_{mn})$
- Umformung auf Basis von Äquivalenzregeln logischer Operatoren

Äquivalenzregeln

- $p_1 \wedge p_2 \Longleftrightarrow p_2 \wedge p_1$ und $p_1 \vee p_2 \Longleftrightarrow p_2 \vee p_1$
- $p_1 \wedge (p_2 \wedge p_3) \Longleftrightarrow (p_1 \wedge p_2) \wedge p_3$ und $p_1 \vee (p_2 \vee p_3) \Longleftrightarrow (p_1 \vee p_2) \vee p_3$
- $p_1 \wedge (p_2 \vee p_3) \Longleftrightarrow (p_1 \wedge p2) \vee (p_1 \wedge p_3)$ und
  $p_1 \vee (p_2 \wedge p_3) \Longleftrightarrow (p_1 \vee p2) \wedge (p_1 \vee p_3)$
- $\neg(p_1 \wedge p_2) \Longleftrightarrow \neg p_1 \vee \neg p_2$ und $\neg(p_1 \vee p_2) \Longleftrightarrow \neg p_1 \wedge \neg p_2$
- $\neg(\neg p_1) \Longleftrightarrow p_1$

Beispiel

SELECT EName
FROM Employees e, Assignment a
WHERE e.EID = a.ENo **AND** Duration $\geq$ 3 **AND** (PNo='P1' **OR** PNo='P2')

Selektionsbedingung in disjunktiver Normalform

$$(\text{EID} = \text{ENo} \ \wedge \ \text{Duration} \geq 3 \ \wedge \ \text{PNo='P1'}) \ \vee$$
$$(\text{EID} = \text{ENo} \ \wedge \ \text{Duration} \geq 3 \ \wedge \ \text{PNo='P2'})$$

Selektionsbedingung in konjunktiver Normalform

$$\text{EID} = \text{ENo} \ \wedge \ \text{Duration} \geq 3 \ \wedge \ (\text{PNo='P1'} \ \vee \ \text{PNo='P2'})$$

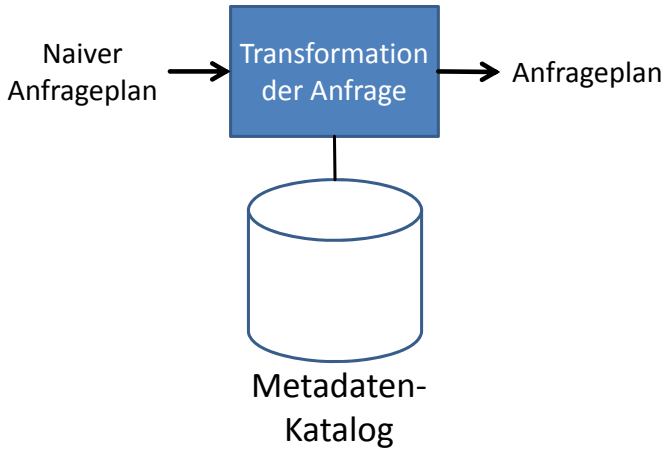
Einfache algebraische Umformungen

Einfache Optimierungen, die immer nützlich sind ungeachtet des aktuellen Systemzustandes

- Eliminierung von redundanten Prädikaten
- Vereinfachung von Ausdrücken
- Herausziehen (un-nesting) von Unteranfragen und Views

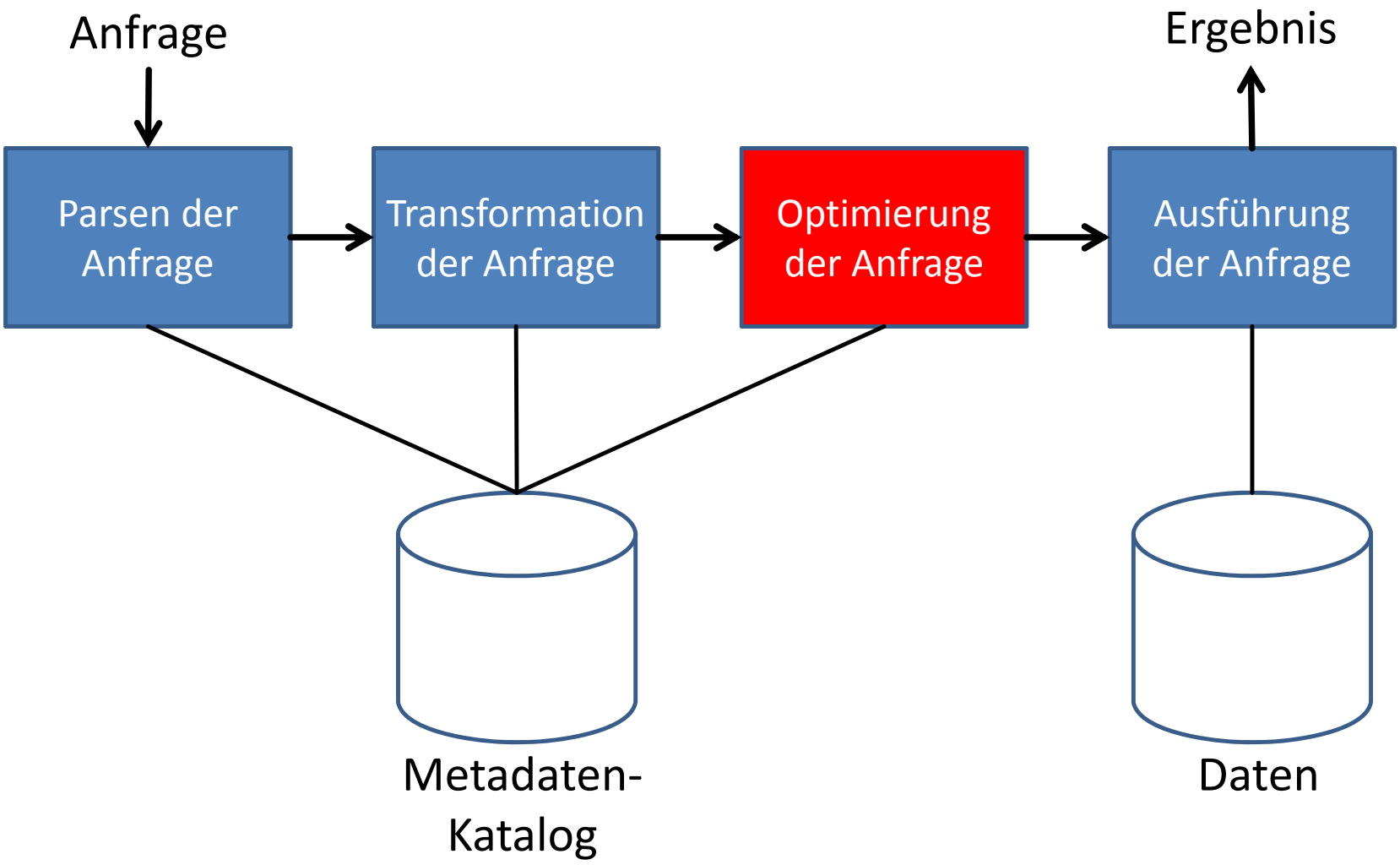
Aufgaben

- Erkennen und **Vereinfachen** aller Ausdrücke/Operationen/Unteranfragen, die “offensichtlich” unnötig, redundant, oder widersprüchlich sind
- Verwendet keine Information über den Systemzustand, z.B. Größe von Tabellen, Existenz von Indexen, etc.



Anfrageoptimierung

Workflow der zentralen Anfrageverarbeitung



Schritte

1. Algebraische Optimierung
   - Finde einen guten äquivalenten algebraischen Operatorbaum
   - Heuristische Anfrageoptimierung
   - Kostenbasierte Anfrageoptimierung
   - Statistische Anfrageoptimierung
2. Physische Optimierung
   - Bestimme geeignete Algorithmen, die die Operationen implementieren

Heuristiken

- Verwende einfache Heuristiken, die gewöhnlich zu **besserer Performance** führen
- Ziel ist nicht unbedingt der beste Plan, aber die wirklich schlechten Pläne sollen vermieden werden
- Heuristiken
  - Zerlege Selektionen
    Komplexe Selekionskriterien sollen in mehrere Teile zerlegt werden
  - Schiebe Projektionen und Selektionen nach unten
    Billige Selektionen und Projektionen sollen so früh wie möglich ausgeführt werden, um die Größe der Zwischenergebnisse zu reduzieren
  - Erzwinge Joins
    In den meisten Fällen ist ein Join viel billiger als ein kartesisches Produkt und eine Selektion

Algebraische Optimierungsregeln

Der $\bowtie$-Operator ist kommutativ:

$$r_1 \bowtie r_2 \iff r_2 \bowtie r_1$$

Der $\bowtie$-Operator ist assoziativ:

$$(r_1 \bowtie r_2) \bowtie r_3 \iff r_1 \bowtie (r_2 \bowtie r_3)$$

Für einen Operator π in Kombination mit einem anderen Operator π dominiert der "äußere" Operator den "inneren" Operator:

$$\pi_X(\pi_Y(r_1)) \iff \pi_X(r_1) \text{ wenn } X \subseteq Y$$

Algebraische Optimierungsregeln

Kombinationen von Selektionen σ können zusammengefasst werden mittels des logischen *und* ($\wedge$). Die Reihenfolge der Selektionen ist beliebig:

$$\sigma_{F_1}(\sigma_{F_2}(r_1)) \Longleftrightarrow \sigma_{F_1 \wedge F_2}(r_1) \Longleftrightarrow \sigma_{F_2}(\sigma_{F_1}(r_1))$$

Ausnutzen der Kommutativität von $\wedge$

Universität Trier

Algebraische Optimierungsregeln

Die Operatoren π und σ kommutieren, wenn das Selektionsprädikat F auf Basis der Projektionsattribute definiert ist:

$$\sigma_F(\pi_X(r_1)) \Longleftrightarrow \pi_X(\sigma_F(r_1)) \text{ wenn } attr(F) \subseteq X$$

Alternativ kann die Reihenfolge vertauscht werden, wenn die Projektion um alle notwendigen Attribute erweitert wird:

$$\pi_{X_1}(\sigma_F(r_1)) \Longleftrightarrow \pi_{X_1}(\sigma_F(\pi_{X_1,X_2}(r_1))) \text{ wenn } attr(F) \subseteq X_2$$

Die Operatoren σ und $\bowtie$ kommutieren, wenn alle Selektionsattribute in der gleichen Relation enthalten sind:

$$\sigma_F(r_1 \bowtie r_2) \Longleftrightarrow \sigma_F(r_1) \bowtie r_2 \text{ wenn } attr(F) \subseteq R_1$$

Ein Selektionsprädikat ($F = F_1 \wedge F_2$) zusammen mit einem Join kann aufgeteilt werden, wenn die Attribute, die durch F_1 und F_2 verwendet werden, in verschiedenen Relationen enthalten sind:

$$\sigma_F(r_1 \bowtie r_2) \Longleftrightarrow \sigma_{F_1}(r_1) \bowtie \sigma_{F_2}(r_2)$$

$$\text{wenn } attr(F_1) \subseteq R_1 \text{ und } attr(F_2) \subseteq R_2$$

In jedem Fall kann ein Teil einer Selektion abgespalten werden, indem Prädikate F_1 abgetrennt werden, die nur Attribute aus R_1 enthalten; F_2 enthält dann die übrigen Prädikate, die Attribute aus beiden Relationen verwenden

$$\sigma_F(r_1 \bowtie r_2) \Longleftrightarrow \sigma_{F_2}(\sigma_{F_1}(r_1) \bowtie r_2) \text{ wenn } attr(F_1) \subseteq R_1$$

Kommutativität von σ und $\cup$:

$$\sigma_F(r_1 \cup r_2) \Longleftrightarrow \sigma_F(r_1) \cup \sigma_F(r_2)$$

Kommutativität von σ und $-$:

$$\sigma_F(r_1 - r_2) \Longleftrightarrow \sigma_F(r_1) - \sigma_F(r_2)$$

oder, falls F nur Tupel in r_1 referenziert:

$$\sigma_F(r_1 - r_2) \Longleftrightarrow \sigma_F(r_1) - r_2$$

Kommutativität von π und $\bowtie$:

$$\pi_X(r_1 \bowtie r_2) \Longleftrightarrow \pi_X(\pi_{Y_1}(r_1) \bowtie \pi_{Y_2}(r_2))$$

mit

$$Y_1 = (X \cap R_1) \cup (R_1 \cap R_2)$$

und

$$Y_2 = (X \cap R_2) \cup (R_1 \cap R_2)$$

Eine Projektion kann nach unten geschoben werden, wenn alle Y_i so definiert sind, dass sie die Attribute erhalten, die für den Join nötig sind.

Weitere Regeln

- Kommutativität von π und $\cup$:

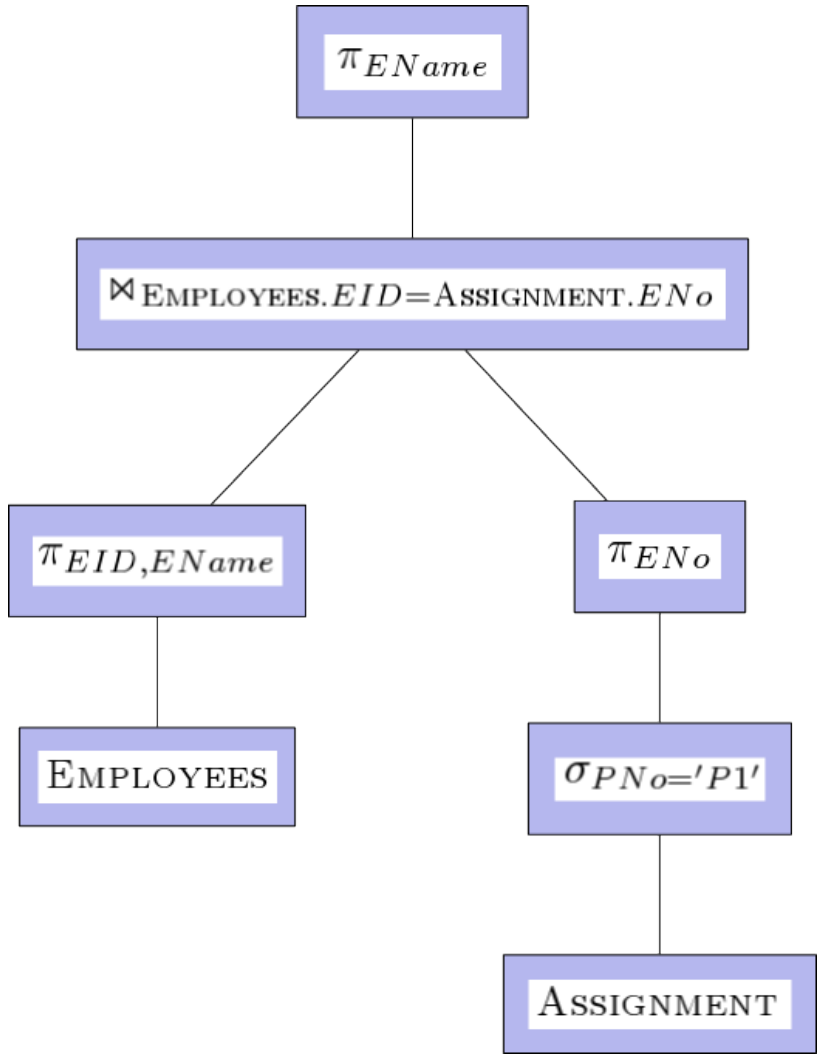
$$\pi_X(r_1 \cup r_2) \Longleftrightarrow \pi_X(r_1) \cup \pi_X(r_2)$$

- Distributivgesetz für $\bowtie$ und $\cup$, Distributivgesetz für $\bowtie$ und $-$, Kommutativität der Umbenennung β mit anderen Operatoren, ...
- Idempotenz, z.B. $A \vee A \Longleftrightarrow A$
- Operationen auf leeren Relationen
- Kommutativ- und Assoziativgesetze für $\bowtie$, $\cup$ und $\cap$

Heuristische algebraische Optimierungen – Beispiel

Verwende algebraische Optimierungsheuristik

- Erzwinge Join
- Schiebe Selektion und Projektion



Kostenbasierte algebraische Anfrageoptimierung

Die meisten nicht-verteilten RDBMS verlassen sich stark auf kostenbasierte Optimierung

- Erstelle besseren optimierten Plan mit Hinblick auf Eigenschaften des Systems und der Daten
  Optimierung der Joinreihenfolge
- Grundlegender Ansatz
  - Erstelle ein Kostenmodell für verschiedene Operationen
  - Zähle alle möglichen Anfragepläne auf und berechne/schätze ihre Kosten
  - Wähle den besten Anfrageplan
- In der Regel verwendet man Dynamisches Programmieren, um den Berechnungsaufwand unter Kontrolle zu halten

Physische Anfrageoptimierung

Physische Optimierung

- Eingabe:
  Optimierter Anfrageplan bestehend aus Algebraoperatoren
- Wähle einen Algorithmus, um jeden Algebraoperator auszuführen
- Join:
  Block-Nested-Loop Join, Hash Join, Merge Join, ...
- Select:
  Kompletter Scan der Tabelle, Nachschlagen im Index, Anlegen eines Index zur Laufzeit und dort nachschlagen, ...

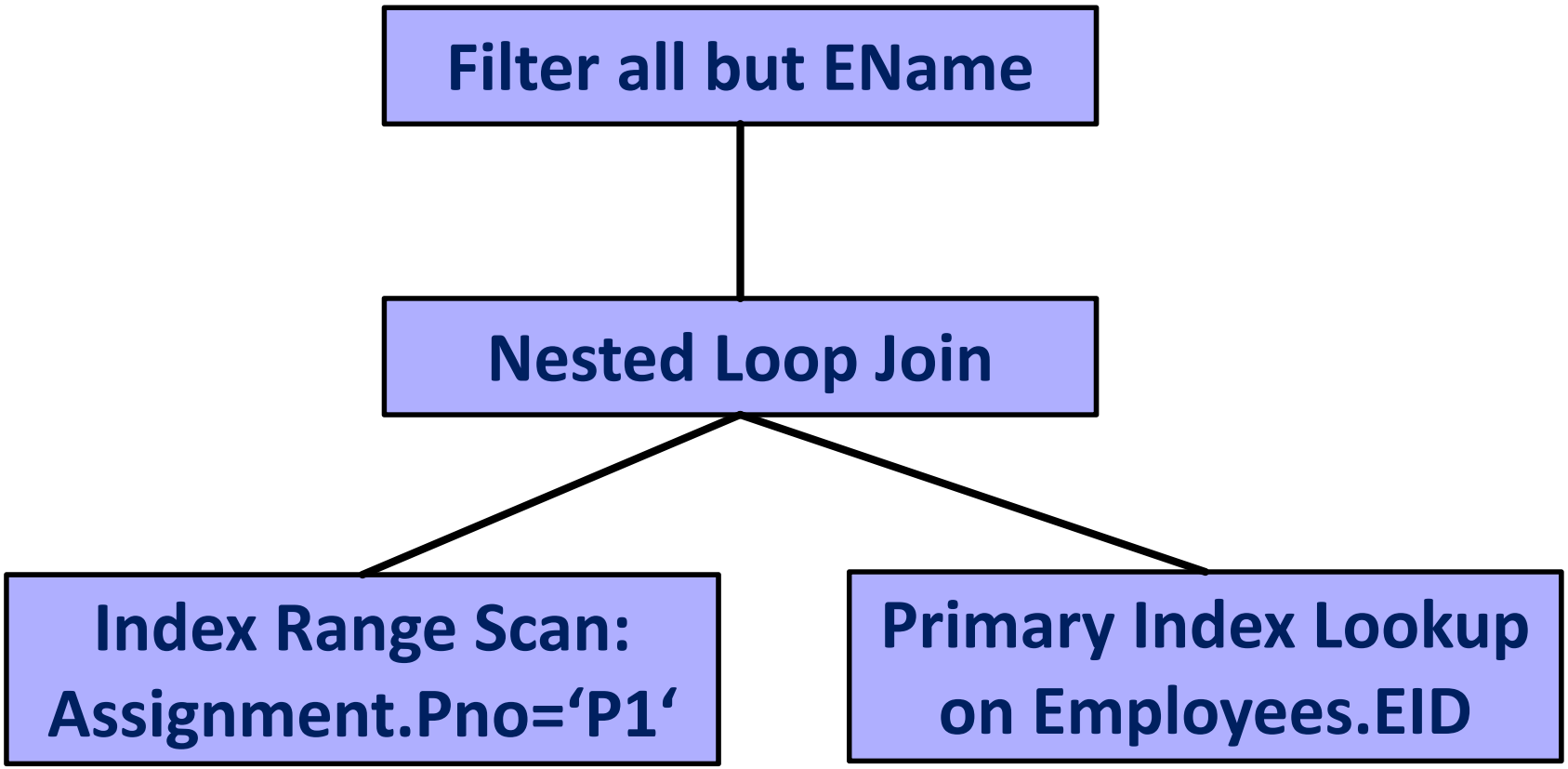
Aufgaben

- Übersetzen eines Anfrageplans in einen Ausführungsplan

Physische und algebraische Optimierung sind oft verschränkt

Beispiel für die Anfrageoptimierung

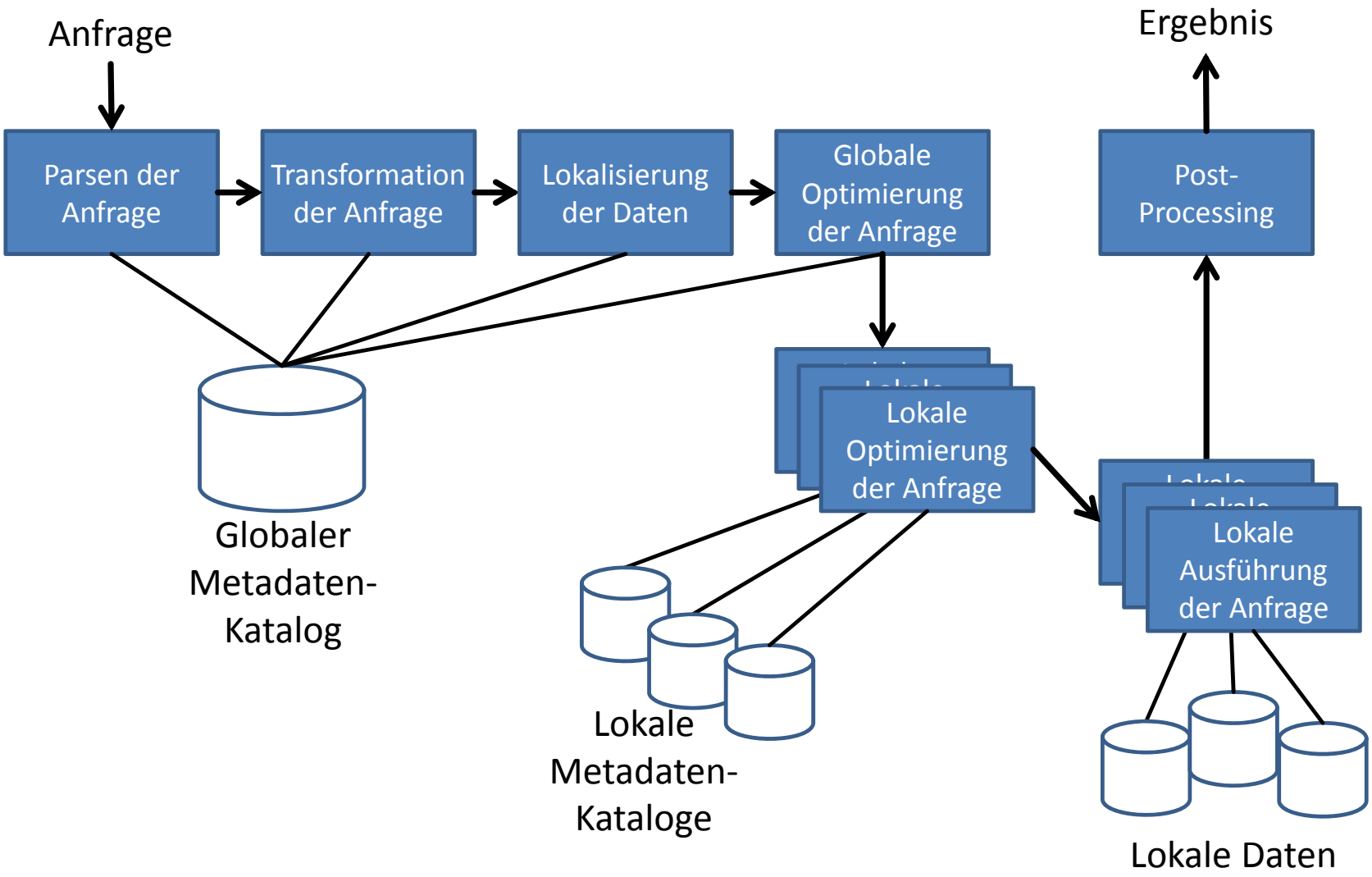
Ausgabe: Ausführungsplan



Grundlagen der verteilten Anfrageausführung

Phasen der verteilten Anfrageausführung

Workflow der verteilten Anfrageausführung



Einführung

Grundlegende Überlegungen

verteilte Anfrageverarbeitung

- Hat viele Gemeinsamkeiten mit zentraler Anfrageverarbeitung
- Ähnliche Probleme, aber andere Ziele und Randbedingungen

Ziele zentraler Anfrageverarbeitung

- Minimiere die Anzahl der Plattenzugriffe
- Minimiere Berechnungszeit

Ziele verteilter Anfrageverarbeitung

- Minimiere Ressourcenverbrauch
- Minimiere Antwortzeit
- Maximiere Durchsatz

Grundlegende Überlegungen

Kosten sind schwieriger vorherzusagen

- Selektivität eines Joins: Ist es sinnvoll, eine Selektion vor dem Join auszuführen?
- Daten sind verteilt: Schwierig, überhaupt sinnvolle Statistiken zu sammeln
- Latenz des Netzes ist sehr schwierig zu bestimmen
- Aktuelle Last auf den Rechnern, Lastabwurf

Zusätzliche Kostenfaktoren und Randbedingungen

- Erweiterung der relationalen Algebra um Senden/Empfangen von Daten
- Datenlokalisierung (welcher Rechner hält relevante Daten)
- Replikation und Caching (wo soll eine Operation ausgeführt werden)
- Modelle für das Verhalten des Netzes
- Modelle für die Antwortzeit
- Heterogenität von Daten und Struktur/Schema (föderierte Datenbanken . . .)

Konsequenzen

Optimierung ist viel schwieriger als im zentralen Fall

- Statistiken und Kosten ändern sich mit der Zeit, z.B. Last auf einem Rechner, Last auf dem Netz
- mehrere, sich widersprechende Optimierungsziele
  Erhöhe Durchsatz $\rightarrow$ *reduziere Replikation und Parallelität,*
  Reduziere Antwortzeit $\rightarrow$ *erhöhe Parallelität*
- Weitere Kostenfaktoren und Randbedingungen

Konsequenzen

- Adaptive Anfragepläne (erzeuge initialen Plan und optimiere ihn während der Ausführung (on-the-fly))
- Generiere nicht den besten Plan, sondern einen guten Plan

Beispiel

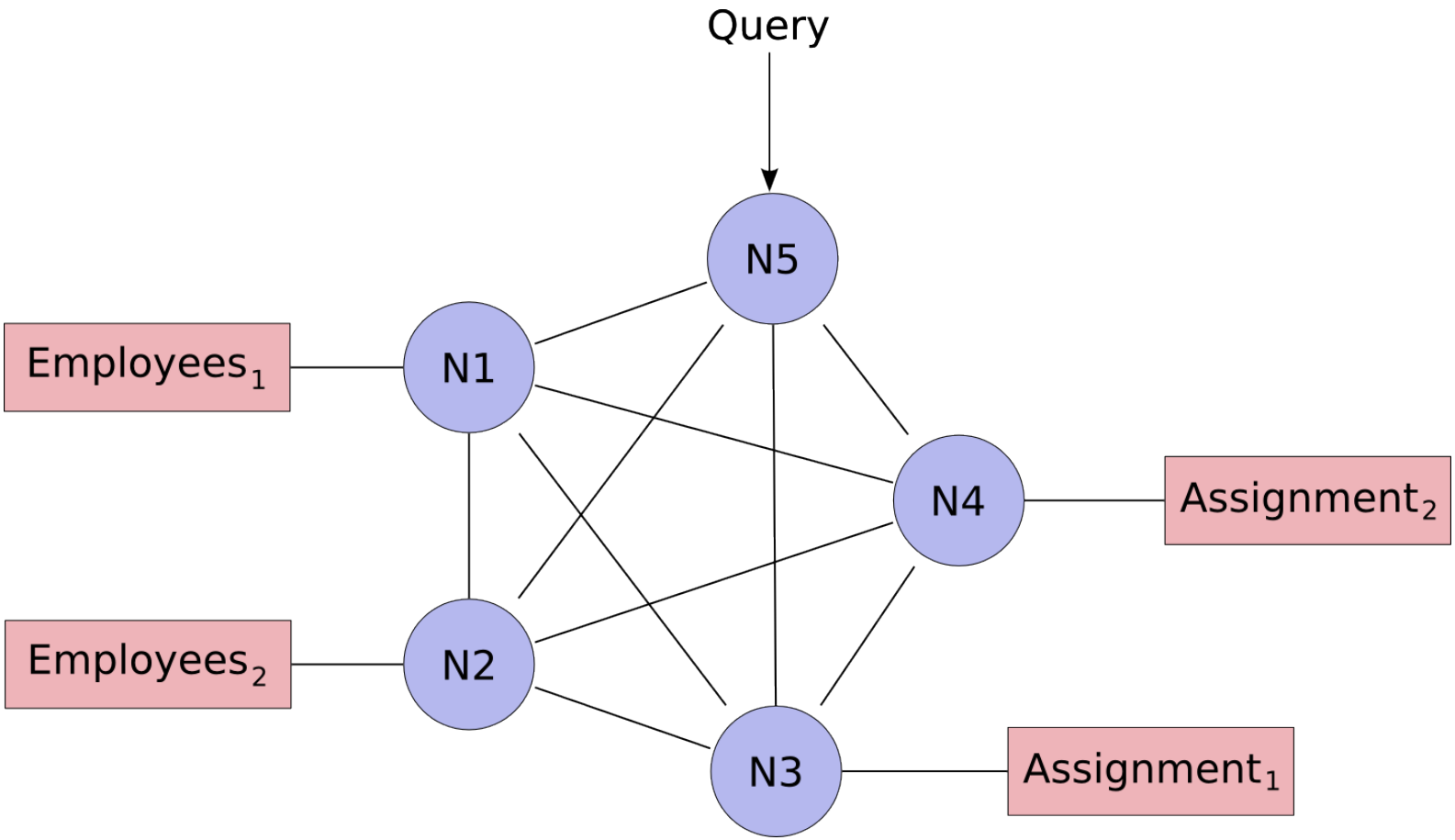
Anfrage

- Gib die Namen aller Angestellten zurück, die im Projekt 'P1' arbeiten
- $\pi_{EName}(\pi_{EID,EName}(\text{EMPLOYEES}) \bowtie_{\text{EMPLOYEES}.EID=\text{ASSIGNMENT}.ENo} \pi_{ENo}(\sigma_{PNo='P1'}(\text{ASSIGNMENT})))$

Probleme

- Relationen sind fragmentiert und über fünf Rechner verteilt
- Die Relation EMPLOYEES verwendet primäre horizontale Fragmentierung
  Ein Fragment auf Rechner 1, das andere auf Rechner 2, keine Replikation
- Die Relation ASSIGNMENT verwendet abgeleitete horizontale Fragmentierung
  Ein Fragment auf Rechner 3, das andere auf Rechner 4, keine Replikation
- Die Anfrage wird auf Rechner 5 gestellt

Beispiel


Query
N5
Employees$_1$
N1
N4
Assignment$_2$
Employees$_2$
N2
N3
Assignment$_1$

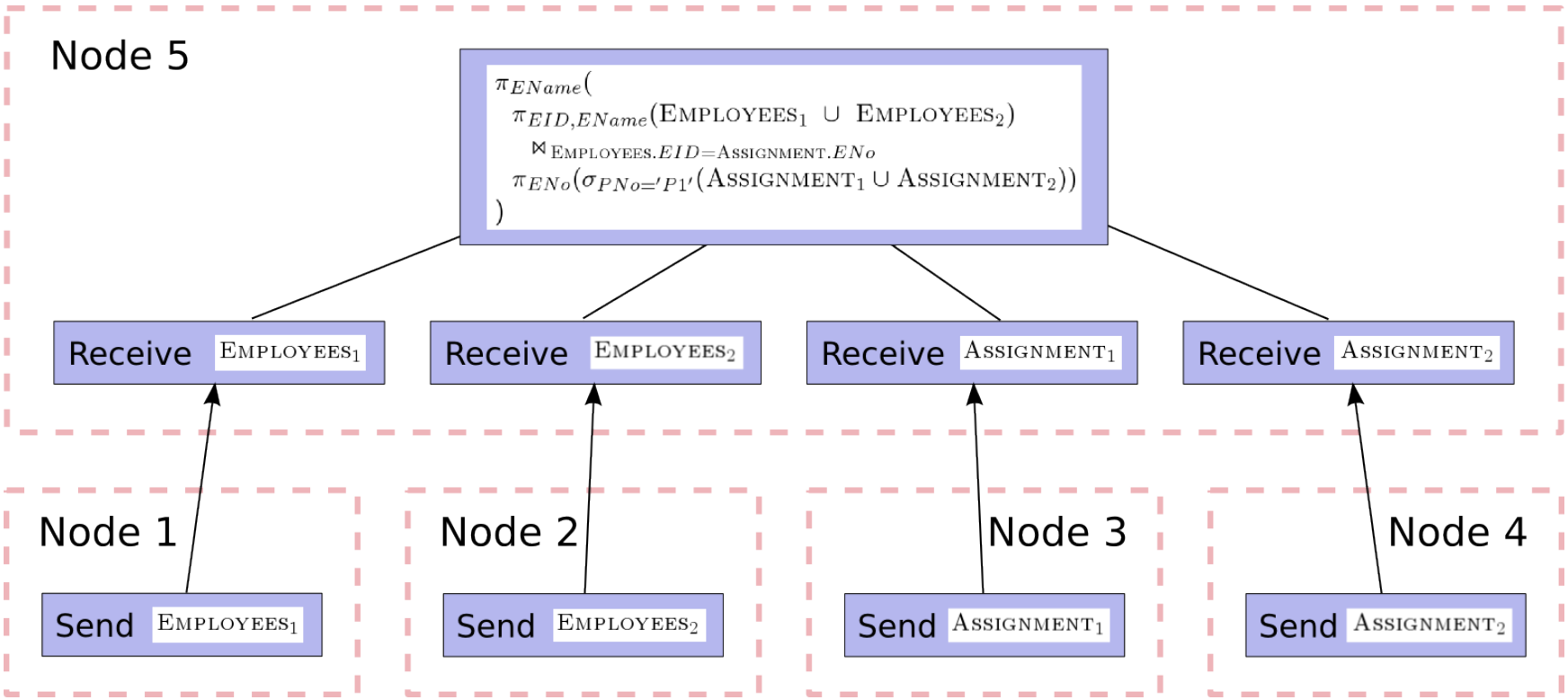
Beispiel

Kostenmodell und Statistiken

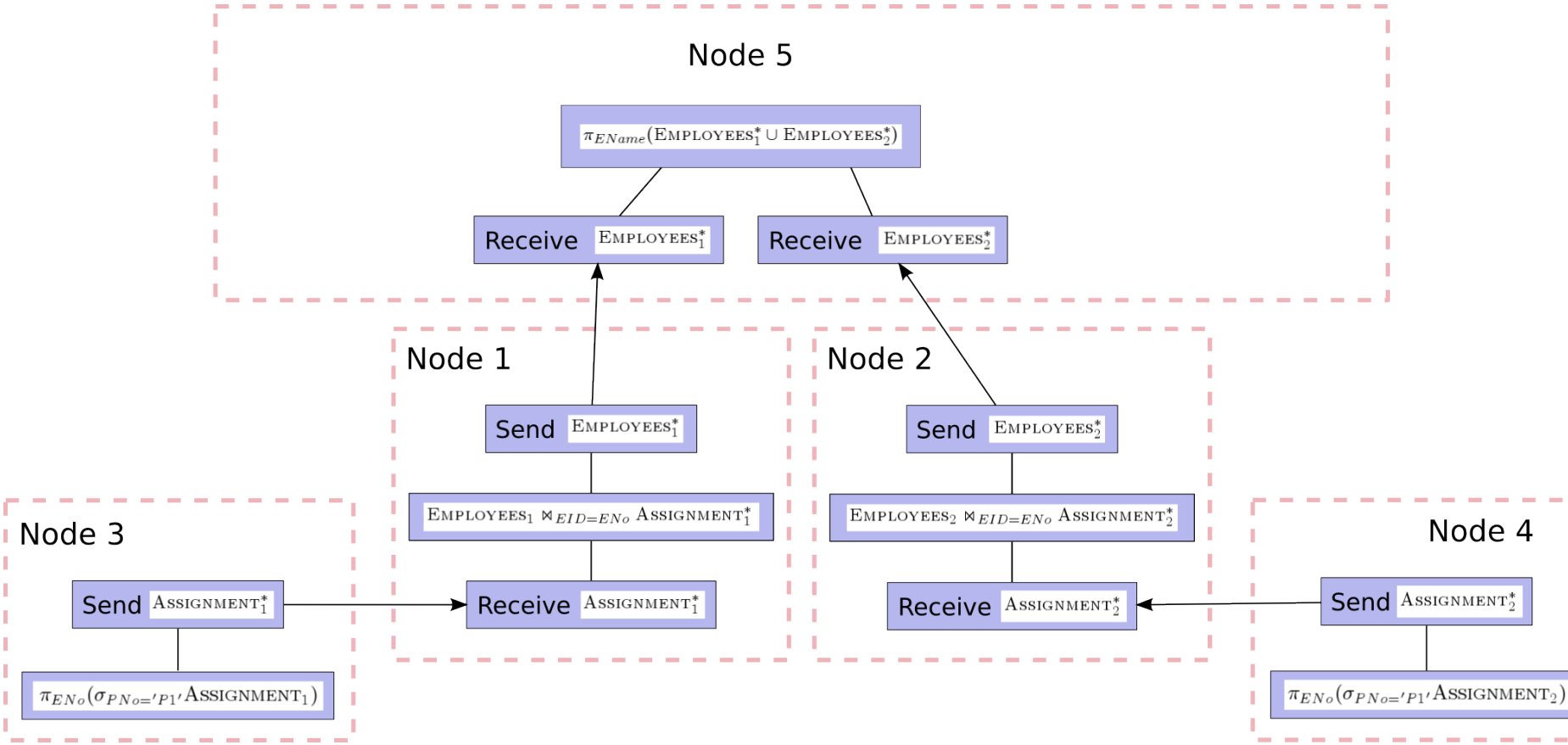
- Zugriff auf ein Tupel hat Kosten 1 (*acc*)
- Übertragen eines Tupels hat Kosten 10 (*trans*)
- Es gibt 400 Tupel in der Relation EMPLOYEES und 1000 Tupel in der Relation ASSIGNMENTS
- Es gibt 20 Assignments für das Projekt 'P1'
- Alle Tupel sind über die jeweiligen Fragmente gleichverteilt, d.h. die Rechner 3 und 4 stellen jeweils 10 Assignments für Projekt 'P1' bereit
- Es gibt lokale Indexe auf Attribut *PNo* auf den Rechnern 3 und 4 (sowie Indexe auf den Primärschlüsseln auf allen Rechnern)
  Direkter Zugriff auf ein Tupel ist überall möglich, kein Scannen von Tabellen erforderlich
- Alle Rechner können direkt miteinander kommunizieren
- Vereinfachung: Keine zusätzlichen Kosten für Projektionen und Vereinigungen

Einfacher Ausführungsplan - Version A

Übertrage alle Daten auf Knoten 5



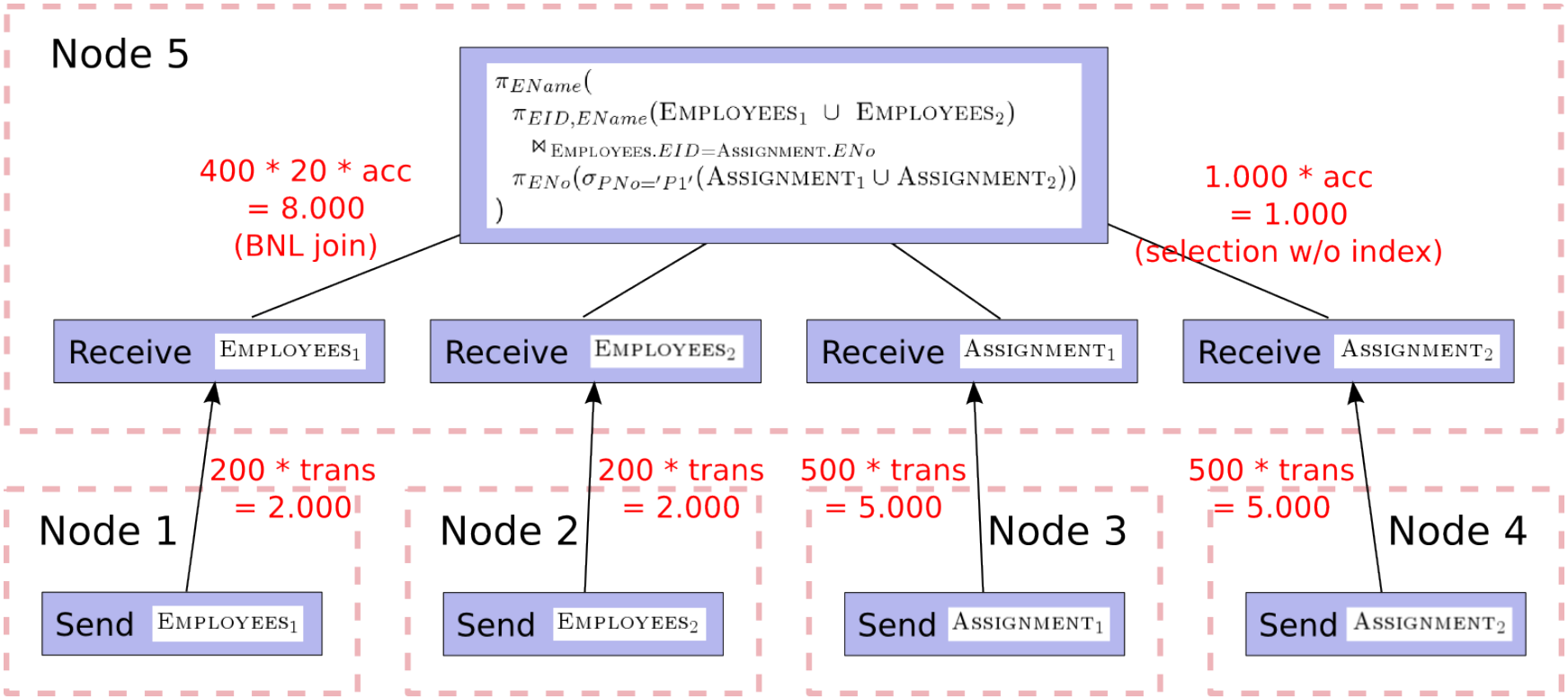
Einfacher Ausführungsplan - Version B
Verschicke Zwischenergebnisse



Universität Trier

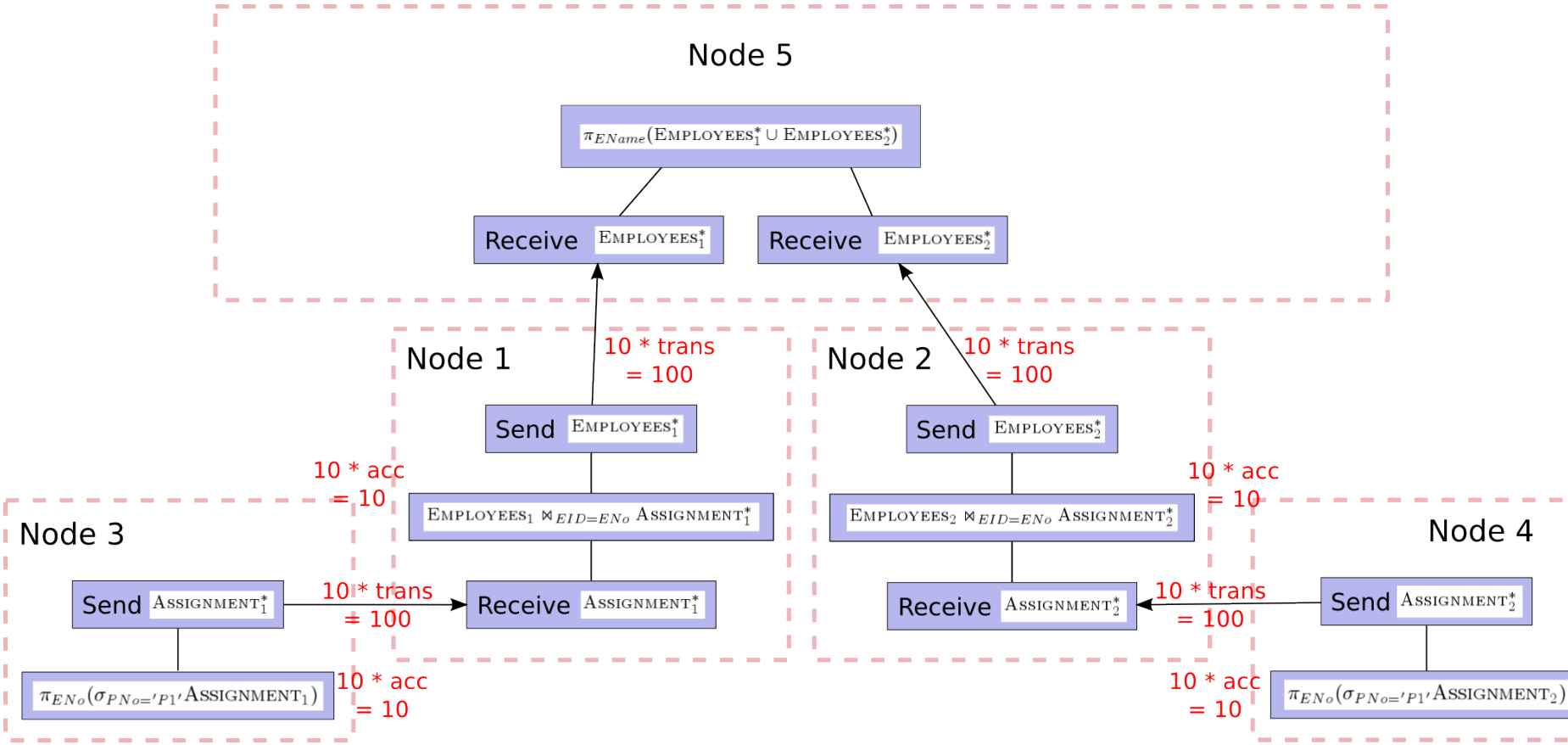
Beispiel

Kosten von Plan A: 23.000



Beispiel

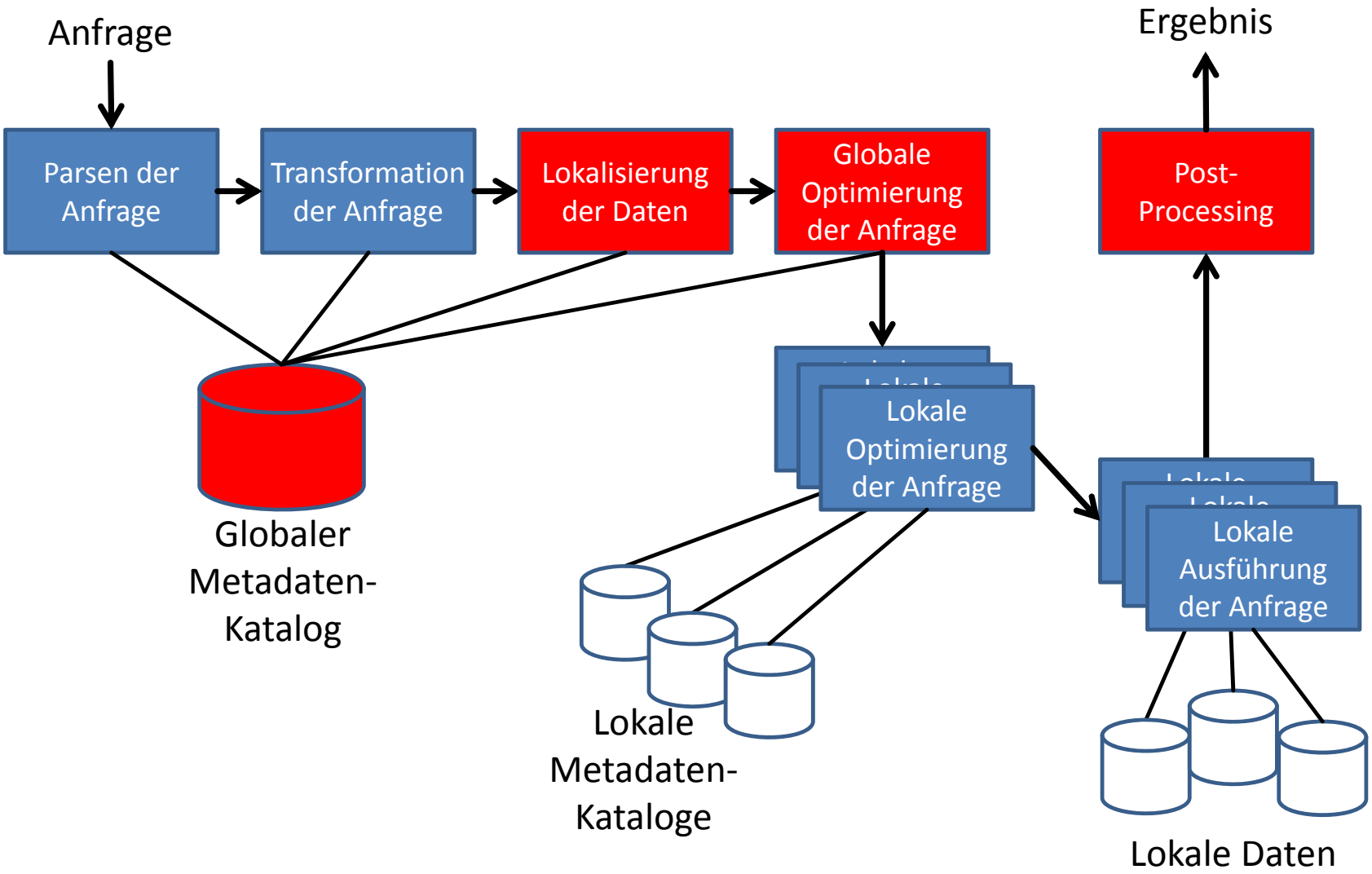
Kosten von Plan B: 440



Wichtige Aspekte verteilter Anfrageverarbeitung

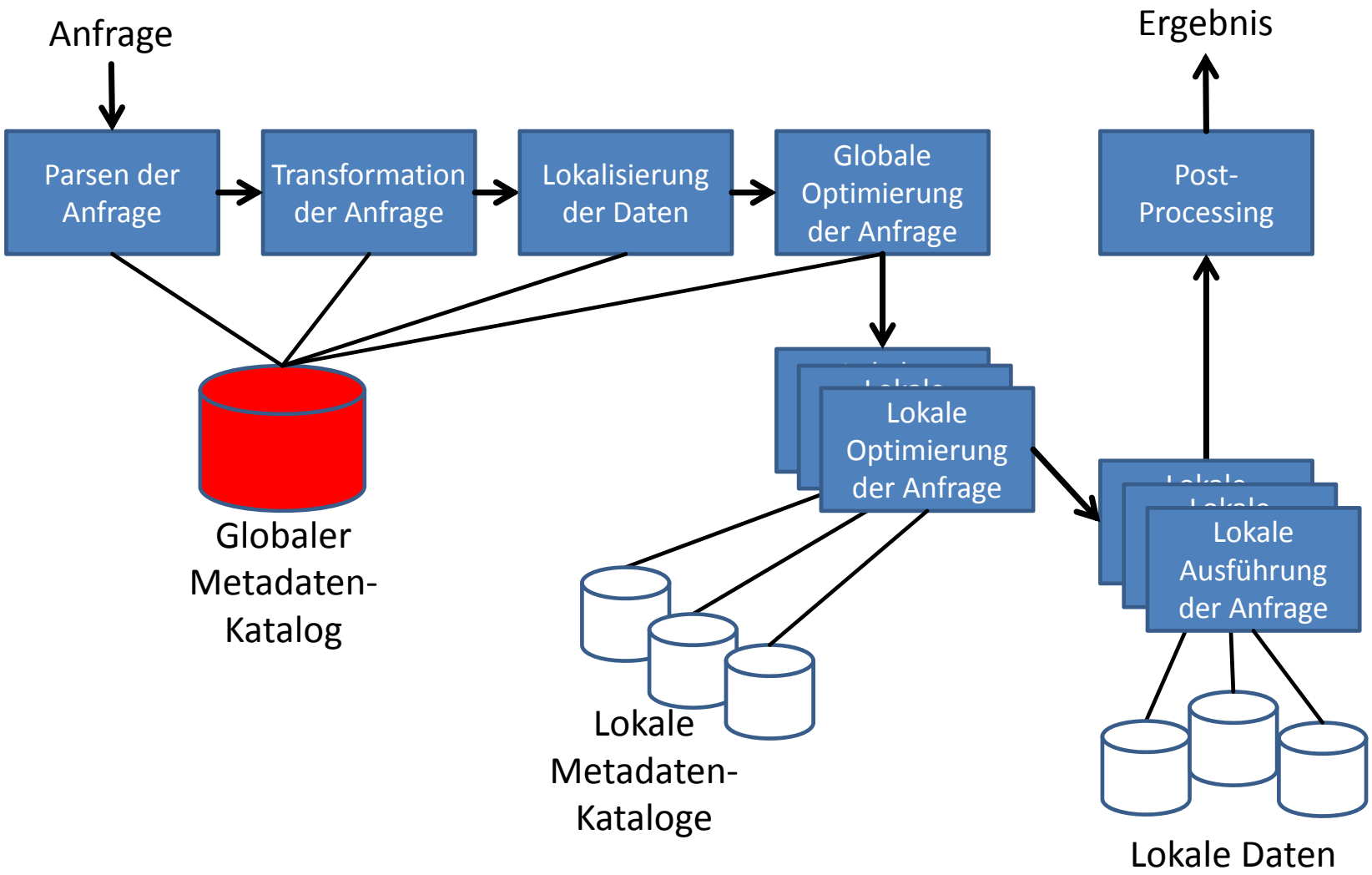
- Metadaten-Management
- Lokalisieren der Daten
- Globale Anfrageoptimierung
- Post-Processing

Wichtige Aspekte verteilter Anfrageverarbeitung



Metadaten-Management

Workflow der verteilten Anfrageausführung



Metadaten-Management

Voraussetzungen für die Anfrageoptimierung

- Metadaten müssen verfügbar sein
- Metadaten werden im Katalog gespeichert
- Katalog stellt Informationen bereit über die Verteilung der (Werte der) Daten

Diese Information wird zum Beispiel verwendet um zu entscheiden, ob es sich lohnt, eine Selektion früh auszuführen.

Typische Inhalte eines Katalogs eines verteilten Datenbanksystems

- Datenbankschema
  Definitionen von Tabellen, Views, Constraints, Schlüsseln, . . .
- Partitionierungsschema
  Information, wie das Schema partitioniert wurde und wie Tabellen rekonstruiert werden können
- Allokationsschema
  Information, welches Fragment auf welchem Rechner gefunden werden kann (einschließlich Information über Replikation)
- Netzinformation
  Information über Verbindungen von Rechnern, Modell des Netzes
- Zusätzliche physische Information
  Information über Indexe, Datenstatistiken (Histogramme etc.), Hardwareressourcen (Verarbeitung & Speicherung),. . .

Metadaten-Management

Wo soll der Katalog in einem verteilten System gespeichert werden?

- zentraler Rechner
  Einfache Lösung, potentieller Engpass
- repliziert auf allen Rechnern
  Änderungen am Katalog sind teuer
- Fragmentiert
  In seltenen Fällen kann der Katalog sehr groß werden
  dann muss der Katalog fragmentiert und alloziert werden
- Caching
  Repliziere nur die benötigten Teile des globalen Katalogs, rechne mit möglichen Inkonsistenzen

Zentraler Katalog

- Eine Instanz des globalen Katalogs auf einem zentralen Rechner
- Vorteile
  - Keine Aktualisierung von Kopien notwendig
  - Wenig Speicherbedarf
- Nachteile
  - Kommunikation mit zentralem Rechner für jede Anfrage
  - Zentraler Rechner ist ein möglicher Engpass

Replizierter Katalog

- Vollständige Kopie des globalen Katalogs auf jedem Rechner
- Vorteile
  - Wenig Kommunikationsaufwand für Anfragen
  - Hohe Verfügbarkeit
- Nachteile
  - Hohe Kosten für Änderungen des Katalogs

Fragmentierung des Katalogs

- Partitionierung des globalen Katalogs und Zuweisung der Partitionen an Rechner
- Vorteile
  - Teilen der Last zwischen den Rechnern
  - Reduzierter Aufwand für Änderungen
- Nachteile
  - Finden der benötigten Partitionen des globalen Katalogs

Cachen von Katalogdaten

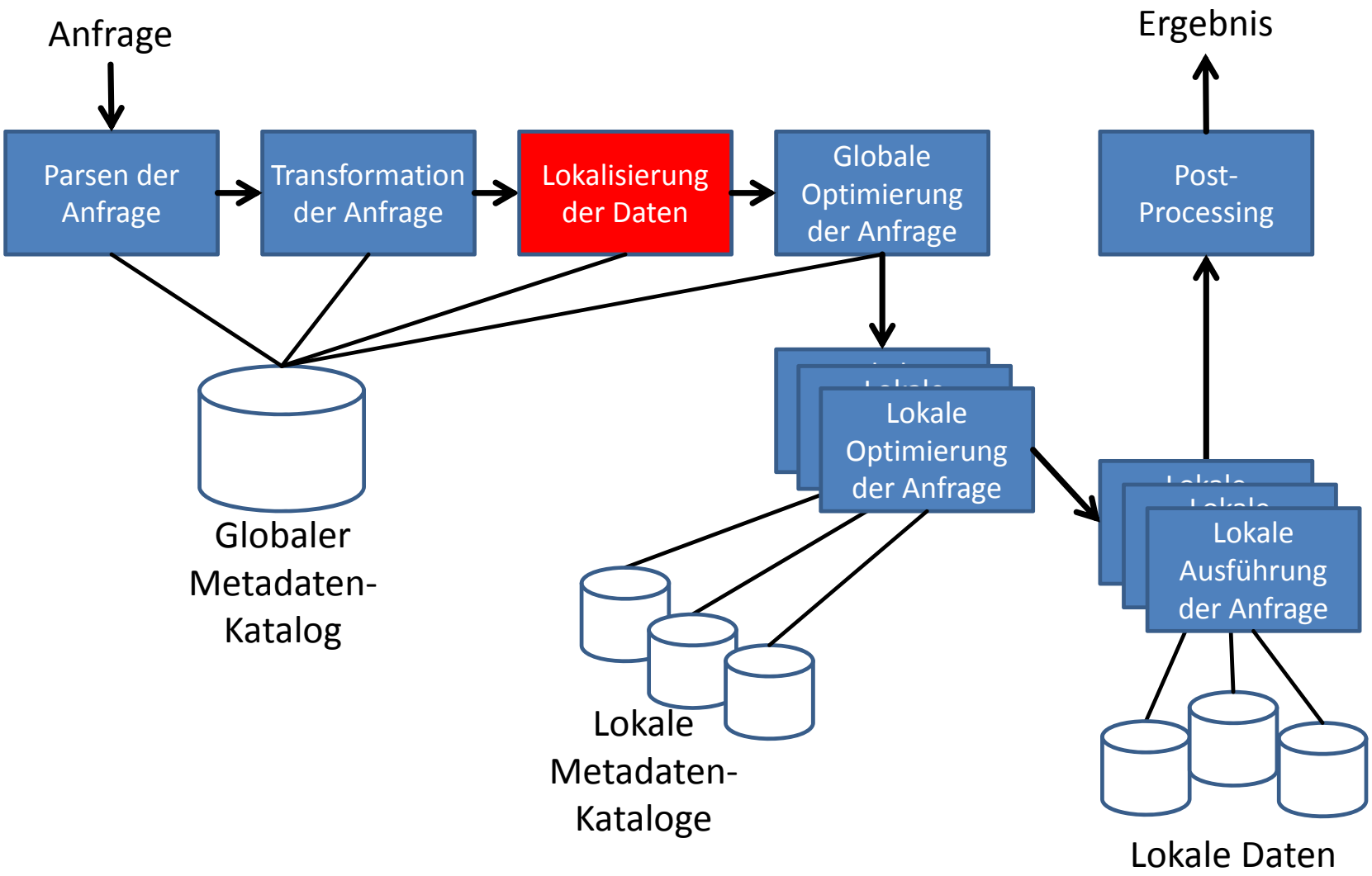
- Cachen nichtlokaler Katalogdaten
- Vorteile
  - Vermeiden des Zugriffs auf einen entfernten Rechner, um häufig benötigte Katalogdaten zu beschaffen
  - Reduzieren des Kommunikationsaufwandes
- Nachteile
  - Kohärenzkontrolle
    Invalidierung gecachter Kopien, wenn Änderungen vorgenommen werden

Cachen von Katalogdaten

- Explizite Invalidierung
  - Besitzer der Katalogdaten merkt sich Rechner mit lokalen Kopien
  - Im Fall einer Änderung: Senden einer Invalidierungsnachricht an alle Rechner mit lokalen Kopien
- Implizite Invalidierung
  - Identifizieren eines veralteten Katalogeintrags zur Laufzeit (Hinzufügen von Versionsnummern und Zeitstempeln zu Anfragenachrichten)

Datenlokalisierung

Workflow der verteilten Anfrageausführung



Datenlokalisierung

Ziel

- Erzeugen von Teilanfragen, die die Verteilung der Daten berücksichtigen

Annahmen

- Fragmentierung wird durch Fragmentierungsausdrücke definiert
- Jedes Fragment ist nur auf einem Rechner alloziert (keine Replikation)
- Fragmentierungsausdrücke und Speicherungsorte der Fragmente sind im Katalog gespeichert

Hauptaufgaben

- Ersetze Zugriff auf globale Relationen durch Zugriffe auf die Fragmente
- Füge Rekonstruktionsausdrücke in die Algebra-Anfrage ein
- Einfache algebraische Vereinfachungen der resultierenden Anfrage

Beispiel – horizontale Rekonstruktion

Schema

- $\text{PROJECTS}_1 = \sigma_{Budget \leq 150.000}(\text{PROJECTS})$
- $\text{PROJECTS}_2 = \sigma_{150.000 < Budget \leq 200.000}(\text{PROJECTS})$
- $\text{PROJECTS}_3 = \sigma_{Budget > 200.000}(\text{PROJECTS})$

Rekonstruktionsausdruck (horizontale Fragmentierung)

- $\text{PROJECTS} = \text{PROJECTS}_1 \cup \text{PROJECTS}_2 \cup \text{PROJECTS}_3$

Beispielanfrage

- $\sigma_{Location='Saarbr.' \wedge Budget \leq 100.000}(\text{PROJECTS})$

Nach dem Ersetzen von globalen Relationen

- $\sigma_{Location='Saarbr.' \wedge Budget \leq 100.000}(\text{PROJECTS}_1 \cup \text{PROJECTS}_2 \cup \text{PROJECTS}_3)$

Weitere Optimierung ist möglich

Ziel

- Eliminiere unnötige Teilanfragen

Horionzontale Reduktionsregel

- Gegeben Fragmente von R als $F_R = \{R_1, \ldots, R_n\}$ mit $R_i = \sigma_{p_i}(R)$
- Alle Fragmente R_i, für die $\sigma_{p_s}(R_i) = \emptyset$, können entfernt werden wobei p_s das Selektionsprädikat der Anfrage angibt
- $\sigma_{p_s}(R_i) = \emptyset \Leftarrow \forall x \in R : \neg(p_s(x) \wedge (p_i(x))$
  Die Selektion mit dem Selektionsprädikat p_s auf Fragment R_i ist leer, wenn p_s dem Fragmentierungsprädikat p_i von R_i widerspricht, d.h. p_s und p_i können niemals gleichzeitig wahr sein für alle Tupel in R_i

Beispiel – horizontale Reduktion

Anfrage mit Fragmentierungsausdruck

$\sigma_{Location='Saarbr.' \wedge Budget \leq 100.000}(\text{PROJECTS}_1 \cup \text{PROJECTS}_2 \cup \text{PROJECTS}_3)$

Definitionen der Fragmente

- $\text{PROJECTS}_1 = \sigma_{Budget \leq 150.000}(\text{PROJECTS})$
- $\text{PROJECTS}_2 = \sigma_{150.000 < Budget \leq 200.000}(\text{PROJECTS})$
- $\text{PROJECTS}_3 = \sigma_{Budget > 200.000}(\text{PROJECTS})$

Weil

$\sigma_{Budget \leq 100.000}(\text{PROJECTS}_2) = \emptyset, \sigma_{Budget \leq 100.000}(\text{PROJECTS}_3) = \emptyset$

erhalten wir die reduzierte Anfrage

$\sigma_{Location='Saarbr.'}(\sigma_{Budget \leq 100.000}(\text{PROJECTS}_1))$

Joinreduktionen

- Joins werden ersetzt durch mehrere Teiljoins auf Fragmenten
- Distributivgesetz: $(R_1 \cup R_2) \bowtie S = (R_1 \bowtie S) \cup (R_2 \bowtie S)$
- Eliminiere alle Union-Fragmente, die leere Ergebnisse zurückliefern werden

Erwartungen

- Eliminierung von Teiljoins, die leere Ergebnisse liefern
  hängt ab von der Güte der Fragmentierung
- Viele Joins auf kleinen Relationen sind billiger als ein großer Join
  hängt ab von der Fragmentierung und den verwendeten Joinalgorithmen
- Kleinere Joins können parallel ausgeführt werden
  könnte Antwortzeit verringern, aber könnte auch Kommunikationskosten erhöhen

Beispiel – Joinreduktion

Schema

PROJECTS(PNo, PName, Budget, Location)

- $\text{PROJECTS}_1 = \sigma_{PNo='P1' \vee PNo='P2'}(\text{PROJECTS})$
- $\text{PROJECTS}_2 = \sigma_{PNo='P3'}(\text{PROJECTS})$
- $\text{PROJECTS}_3 = \sigma_{PNo='P4'}(\text{PROJECTS})$

ASSIGNMENT(ENo, PNo, Duration)

- $\text{ASSIGNMENT}_1 = \sigma_{PNo='P1' \vee PNo='P2'}(\text{ASSIGNMENT})$
- $\text{ASSIGNMENT}_2 = \sigma_{PNo='P3' \vee PNo='P4'}(\text{ASSIGNMENT})$

Beispielanfrage

select * **from** Projects p, Assignment a **where** p.PNo = a.PNo

In relationaler Algebra

$$\text{PROJECTS} \bowtie \text{ASSIGNMENT}$$

Beispiel – Joinreduktion

Anfrage

$$\textsc{Projects} \bowtie \textsc{Assignment}$$

Nach dem Ersetzen globaler Relationen durch Rekonstruktionsausdrücke

$$(\textsc{Projects}_1 \cup \textsc{Projects}_2 \cup \textsc{Projects}_3) \bowtie (\textsc{Assignment}_1 \cup \textsc{Assignment}_2)$$

Nach dem Anwenden des Distributivgesetzes

$$\begin{aligned}&(\textsc{Projects}_1 \bowtie \textsc{Assignment}_1) \cup (\textsc{Projects}_1 \bowtie \textsc{Assignment}_2) \cup \\ &(\textsc{Projects}_2 \bowtie \textsc{Assignment}_1) \cup (\textsc{Projects}_2 \bowtie \textsc{Assignment}_2) \cup \\ &(\textsc{Projects}_3 \bowtie \textsc{Assignment}_1) \cup (\textsc{Projects}_3 \bowtie \textsc{Assignment}_2)\end{aligned}$$

Weitere Optimierung ist möglich

Joinreduktions-Regel

- Gegeben Fragmente von R als $F_R = \{R_1, \ldots, R_n\}$ und Fragmente von S als $F_S = \{S_1, \ldots, S_n\}$
- Wende das Distributivgesetz an, d.h.:
  $(R_1 \cup R_2) \bowtie (S_1 \cup S_2) = (R_1 \bowtie S_1) \cup (R_1 \bowtie S_2) \cup (R_2 \bowtie S_1) \cup (R_2 \bowtie S_2)$
- Alle Teiljoins zwischen Fragmenten R_i und S_j, für die $R_i \bowtie S_j = \emptyset$ gilt, können entfernt werden
- $R_i \bowtie S_j = \emptyset \Leftarrow \forall x \in R_i, y \in S_j : \neg(p_i(x) \wedge p_j(y))$
  Der Join zwischen Fragmenten R_i und S_j ist leer, wenn ihre Fragmentierungsprädikate (auf dem Joinattribut) sich widersprechen, d.h. wenn es keine Kombination von Tupeln x und y geben kann, so dass beide Partitionierungsprädikate zur gleichen Zeit erfüllt sind.

Beispiel – Joinreduktion

Anfrage mit Fragmentierungsausdruck

$$(\text{PROJECTS}_1 \bowtie \text{ASSIGNMENT}_1) \cup (\text{PROJECTS}_1 \bowtie \text{ASSIGNMENT}_2) \cup$$
$$(\text{PROJECTS}_2 \bowtie \text{ASSIGNMENT}_1) \cup (\text{PROJECTS}_2 \bowtie \text{ASSIGNMENT}_2) \cup$$
$$(\text{PROJECTS}_3 \bowtie \text{ASSIGNMENT}_1) \cup (\text{PROJECTS}_3 \bowtie \text{ASSIGNMENT}_2)$$

Einige dieser Teiljoins sind leer, z.B.

$$\text{PROJECTS}_1 \bowtie \text{ASSIGNMENT}_2 = \emptyset$$

weil sich ihre Fragmentierungsausdrücke widersprechen:

$$\text{PROJECTS}_1 = \sigma_{PNo='P1' \vee PNo='P2'}(\text{PROJECTS}) \text{ und}$$
$$\text{ASSIGNMENT}_2 = \sigma_{PNo='P3' \vee PNo='P4'}(\text{ASSIGNMENT})$$

Reduzierte Anfrage

$$(\text{PROJECTS}_1 \bowtie \text{ASSIGNMENT}_1) \cup (\text{PROJECTS}_2 \bowtie \text{ASSIGNMENT}_2) \cup$$
$$(\text{PROJECTS}_3 \bowtie \text{ASSIGNMENT}_2)$$

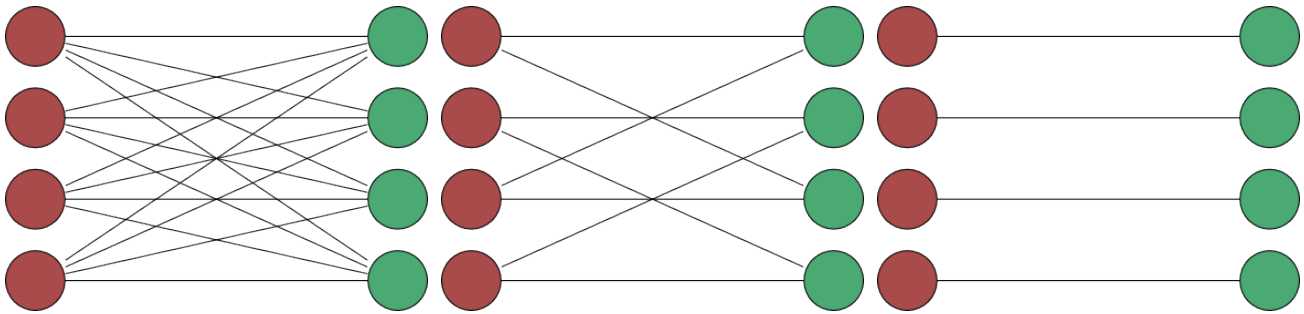
Universität Trier

Anfragevereinfachung - Joinreduktion für horizontale Fragmentierung

Der einfachste Fall einer Joinreduktion ergibt sich bei abgeleiteter horizontaler Fragmentierung

- Für jedes Fragment der ersten Relation gibt es genau ein passendes Fragment der zweiten Relation
- Wir verwenden einfach die Information in den Rekonstruktionsausdrücken, anstatt die Rekonstruktionsprädikate miteinander zu vergleichen

Joinreduktion für beliebige horizontale Partitionierungen ist nicht immer nützlich



Anfragevereinfachung – Joinreduktion für abgeleitete horizontale Fragmentierung

Beispiel

$$\textsc{Projects}(\underline{\text{PNo}}, \text{PName}, \text{Budget}, \text{Location})$$

$$\textsc{Projects}_1 = \sigma_{PNo='P1' \vee PNo='P2'}(\textsc{Projects})$$

$$\textsc{Projects}_2 = \sigma_{PNo='P3' \vee PNo='P4'}(\textsc{Projects})$$

$$\textsc{Assignment}(\underline{\text{ENo}}, \underline{\text{PNo}}, \text{Duration})$$

$$\textsc{Assignment}_1 = \textsc{Assignment} \ltimes \textsc{Projects}_1$$

$$\textsc{Assignment}_2 = \textsc{Assignment} \ltimes \textsc{Projects}_2$$

Anfrage in relationaler Algebra

$$\textsc{Projects} \bowtie \textsc{Assignment}$$

Anfragevereinfachung – Joinreduktion für abgeleitete horizontale Fragmentierung

Nach dem Ersetzen der globalen Relationen durch ihre Rekonstruktionsausdrücke

$$(\text{PROJECTS}_1 \cup \text{PROJECTS}_2) \bowtie (\text{ASSIGNMENT}_1 \cup \text{ASSIGNMENT}_2)$$

Nach dem Anwenden des Distributivgesetzes

$$(\text{PROJECTS}_1 \bowtie \text{ASSIGNMENT}_1) \cup (\text{PROJECTS}_1 \bowtie \text{ASSIGNMENT}_2) \cup$$
$$(\text{PROJECTS}_2 \bowtie \text{ASSIGNMENT}_1) \cup (\text{PROJECTS}_2 \bowtie \text{ASSIGNMENT}_2)$$

Reduzierte Anfrage (unter direkter Verwendung der Information über die Fragmentierung der Relation ASSIGNMENT)

$$(\text{PROJECTS}_1 \bowtie \text{ASSIGNMENT}_1) \cup (\text{PROJECTS}_2 \bowtie \text{ASSIGNMENT}_2)$$

Vertikale Joinreduktionsregel

- Gegeben Fragmente von R als $F_R = \{R_1, \ldots, R_n\}$ mit $R_i = \pi_{\beta_i}(R)$ wobei β_i die Aufzählung einer Teilmenge der Attribute von R repräsentiert
- Vermeide das Joinen von Fragmenten, die "nutzlose" Attribute enthalten, d.h. Fragmente, die nur Attribute enthalten, die nicht in der Anfrage verwendet werden und nicht im Ergebnis enthalten sind.

Beispiel – vertikale Reduktion

Schema

PROJECTS(PNo, PName, Budget, Location)

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

Rekonstruktionsausdruck

- $\text{PROJECTS} = \text{PROJECTS}_1 \bowtie \text{PROJECTS}_2$

Beispielanfrage

- $\pi_{PName}(\text{PROJECTS})$

Nach dem Ersetzen der globalen Relationen

- $\pi_{PName}(\text{PROJECTS}_1 \bowtie \text{PROJECTS}_2)$

Nach dem Entfernen unnötiger Fragmente

- $\pi_{PName}(\text{PROJECTS}_1)$

Anfragevereinfachung – hybride Fragmentierung

- Der Rekonstruktionsausdruck verwendet Kombinationen von Joins und Vereinigungen
- Allgemeine Regeln
  - Entferne leere Relationen, die durch sich widersprechende Prädikate auf horizontalen Fragmenten entstanden sind
  - Entferne nutzlose Relationen, die durch vertikale Fragmente entstanden sind
  - Zerteile und verteile Joins, eliminiere leere Joins von Fragmenten

- Unterstützen von algebraischer Optimierung von Anfragen, die Fragmente enthalten
- Annotieren von Fragmenten und Zwischenergebnissen mit Prädikaten
- Schätzen der Größe einer Relation
- Erweiterung der relationalen Algebra

Definition 4.1 (qualifizierte Relation)

Eine qualifizierte Relation ist ein Paar $[R : q_R]$, wobei R eine Relation ist und q_R ein Prädikat, das jedes Tupel in R erfüllt.

Beispiel 4.2

Wenn wir **horizontale Fragmente** als qualifizierte Relationen darstellen, bei denen das **Qualifizierungsprädikat** dem Fragmentierungsausdruck entspricht, erhalten wir

$$[\text{PROJECTS}_1 : \sigma_{PNo='P1' \vee PNo='P2'}]$$

Erweiterte relationale Algebra

$$
\begin{array}{lll}
(1) & E := & \sigma_F[R : q_R] \rightarrow [E : F \wedge q_R] \\
(2) & E := & \pi_A[R : q_R] \rightarrow [E : q_R] \\
(3) & E := & [R : q_R] \times [S : q_S] \rightarrow [E : q_R \wedge q_S] \\
(4) & E := & [R : q_R] - [S : q_S] \rightarrow [E : q_R] \\
(5) & E := & [R : q_R] \cup [S : q_S] \rightarrow [E : q_R \vee q_S] \\
(6) & E := & [R : q_R] \bowtie_F [S : q_S] \rightarrow [E : q_R \wedge q_S \wedge F]
\end{array}
$$

Beispielanfrage

$$\sigma_{100.000 \leq \text{Budget} \leq 200.000}(\text{PROJECTS})$$

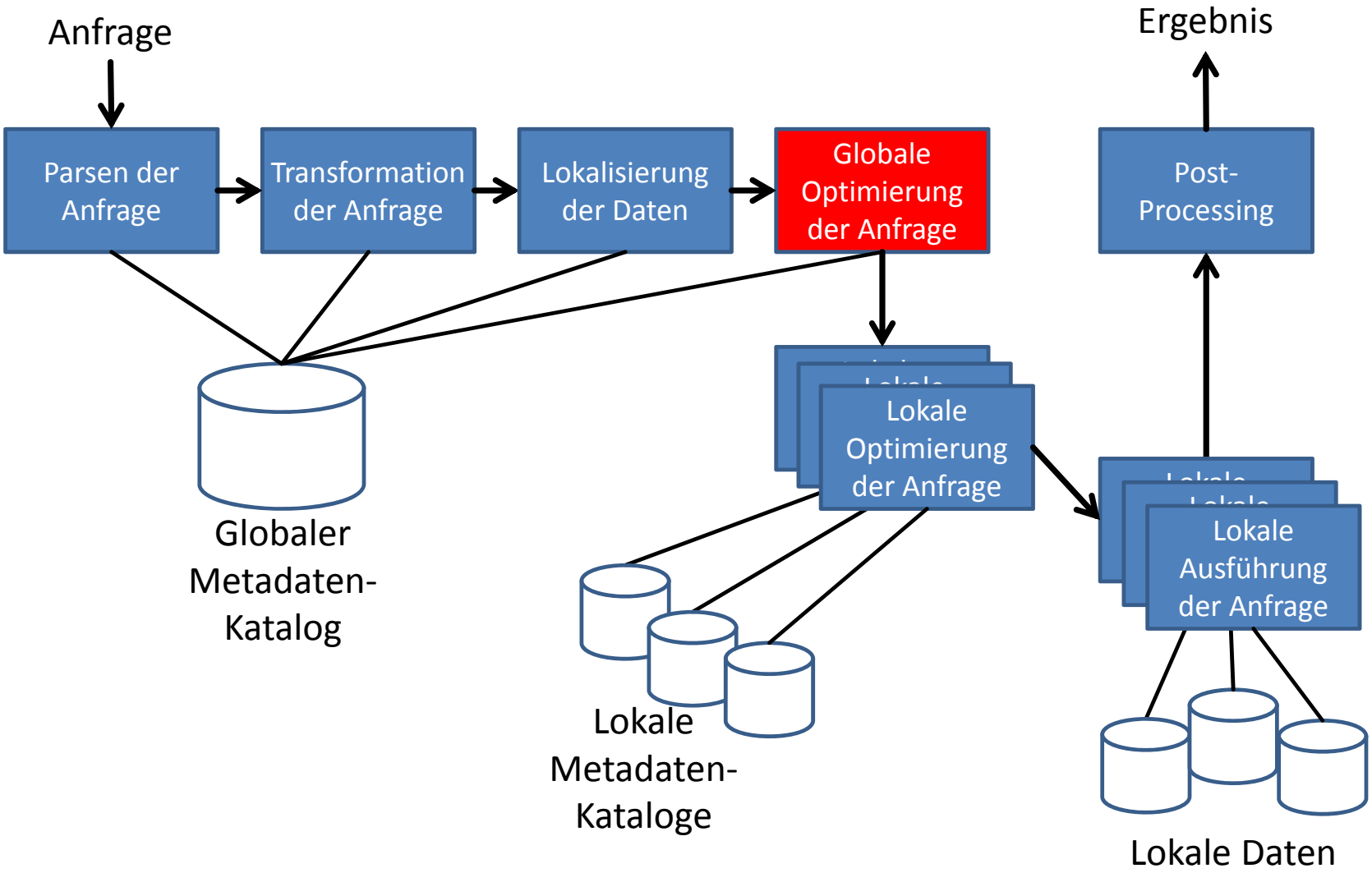
Qualifizierte Relationen

$$
\begin{aligned}
E_1 &= \sigma_{100.000 \leq \text{Budget} \leq 200.000}[\text{PROJECTS}_1 : \text{Budget} \leq 150.000] \\
&\rightsquigarrow [E_1 : (100.000 \leq \text{Budget} \leq 200.000) \wedge (\text{Budget} \leq 150.000)] \\
&\rightsquigarrow [E_1 : 100.000 \leq \text{Budget} \leq 150.000] \\
E_2 &= \sigma_{100.000 \leq \text{Budget} \leq 200.000}[\text{PROJECTS}_2 : 150.000 < \text{Budget} \leq 200.000] \\
&\rightsquigarrow [E_2 : (100.000 \leq \text{Budget} \leq 200.000) \wedge \\
&\qquad (150.000 < \text{Budget} \leq 200.000)] \\
&\rightsquigarrow [E_2 : 150.000 < \text{Budget} \leq 200.000] \\
E_3 &= \sigma_{100.000 \leq \text{Budget} \leq 200.000}[\text{PROJECTS}_3 : \text{Budget} > 200.000] \\
&\rightsquigarrow [E_3 : (100.000 \leq \text{Budget} \leq 200.000) \wedge (\text{Budget} > 200.000)] \\
&\rightsquigarrow E_3 = \emptyset
\end{aligned}
$$

Globale Anfrageoptimierung

Wesentliche Fragen

Workflow der verteilten Anfrageausführung



Wesentliche Fragen

- Wann soll optimiert werden?
- Welche Kriterien sollen optimiert werden?
- Wo soll die Anfrage ausgeführt werden?

Wann soll optimiert werden?

Vollständige Optimierung zur Compilezeit

- Der vollständige Ausführungsplan wird zur Compilezeit berechnet
- Annahme
  - Anwendungen verwenden immer die gleichen Anfragemuster
    Vorbereitete und parametrisierte SQL-Ausdrücke
- Vorteile
  - Anfragen können sofort ausgeführt werden
- Nachteile
  - Modellierung sehr komplex
  - Großer Teil der benötigten Information nicht verfügbar oder zu teuer zu bestimmen
    Einsammeln der Statistiken von allen Rechnern?
  - Statistiken veraltet
    besonders Last der Rechner und Eigenschaften des Netzes sind sehr variabel

Wann soll optimiert werden?

Vollständig dynamische Optimierung

- Jede Anfrage wird individuell zur Laufzeit optimiert
- Diese Technik verlässt sich stark auf Heuristiken, Lernverfahren und Glück
- Vorteile
  - Kann sehr gute Pläne generieren
  - Berücksichtigt den aktuellen Zustand des Netzes
  - Verwendbar für Adhoc-Anfragen
- Nachteile
  - Qualität der resultierenden Pläne unvorhersagbar
  - Komplexe Algorithmen und Statistiken
  - Statistiken aktuell zu halten ist schwierig

Wann soll optimiert werden?

Halb-dynamische Optimierung

- Anfrage wird zur Compilezeit vor-optimiert
- Zur Laufzeit wird überprüft, ob die Ausführung abläuft, wie während der Optimierung geschätzt wurde
  werden Tupel/Fragmente rechtzeitig geliefert? Verhält sich das Netz wie vorhergesagt? Gibt es unerwartete Netzlatenzen? etc.
- Wenn die Ausführung deutlich von der erwarteten abweicht, berechne neuen Ausführungsplan für alle noch nicht ausgeführten Teile der Anfrage

Nur sinnvoll für Anfragen, die länger laufen

Wann soll optimiert werden?

Hierarchische Optimierung

- Pläne werden in mehreren Schritten erstellt
- Global-Lokale Pläne
  - Der globale Anfrageoptimierer erstellt einen globalen Anfrageplan
    Fokus auf Transfer von Daten: Welche Zwischenergebnisse sollen von welchem Rechner berechnet werden? Wie sollen Zwischenergebnisse ausgetauscht werden?
  - Lokale Anfrageoptimierer erstellen lokale Anfragepläne
    Entscheiden über die Struktur des Plans, Algorithmen, Indexe etc., um die verlangten Zwischenergebnisse zu berechnen
- Zwei-Schritt-Pläne

Wann soll optimiert werden?

Hierarchische Optimierung

- Pläne werden in mehreren Schritten erstellt
- Global-Lokale Pläne
- Zwei-Schritt-Pläne
  - Zur Compilezeit werden nur die stabilen Teile des Plans berechnet
    Joinreihenfolge, Joinmethoden, Zugriffspfade, etc.
  - Während der Anfrageausführung werden die fehlenden Teile des Plans hinzugefügt
    Rechnerauswahl, Transfer von Zwischenergebnissen, etc.
  - Beide Schritte können mit traditionellen Optimierungsmethoden gelöst werden
    - Aufzählen möglicher Pläne mit dynamischer Programmierung
    - Komplexität ist kontrollierbar, da jedes Optimierungsproblem einfacher als eine komplette Optimierung ist
    - Während der Laufzeitoptimierung sind aktuelle Statistiken verfügbar

Die meisten verteilten Datenbanksysteme verwenden halbdynamische oder hierarchische Optimierungsverfahren (oder beide)

Welche Kritierien sollen optimiert werden?

Wichtige Aspekte der globalen Optimierung

- Kommunikationsoperatoren
- Fragmentgrößen
- Reihenfolge der Operationen
- Joinreihenfolge
  Weil Vertauschungen der Joins innerhalb der Anfrage zu Verbesserungen von mehreren Größenordnungen führen können

Wichtigste alternative Optimierungskriterien

- Antwortzeit der Anfrage
- Ressourcenverbrauch
- Gesamtausführungskosten auf allen Rechnern
- ...

Universität Trier

Wo soll die Anfrage ausgeführt werden?

- Der Anfrageoptimierer muss entscheiden, welche Teile der Anfrage an welchen Rechner geschickt werden sollen (Kostenmodell)
- In Szenarien mit starker Replikation kann ein intelligentes Verschicken zur Lastbalancierung beitragen
  Verschiebe teure Berechnungen auf schwach ausgelastete Rechner, vermeide teure Kommunikation

Globale Anfrageoptimierung . . .

. . . kümmert sich um die Auswahl der “besten” Anordnung der Operationen in der Anfrage (erweitert um Fragmentierungsausdrücke und Kommunikationsoperationen), die eine Kostenfunktion minimiert.

- Eingabe
  Eine Anfrage in der Algebra, erweitert um Fragmentierungsausdrücke
- Ausgabe
  Eine Anfrage in der Algebra oder ein Ausführungsplan mit Kommunikationsoperationen

Globaler Anfrageoptimierer

Grundlage globaler Anfrageoptimierung

Ziel

- Auswahl eines kosteneffizienten Ausführungsplans basierend auf dem algebraischen Anfrageplan aus der Eingabe
- Entscheidung, welche Teile der Anfrage auf welchem Rechner ausgeführt werden

Voraussetzungen

- Wissen über die Fragmentierung
- Wissen über die Größe von Fragmenten und Relationen
- Wissen über die Verteilung der Daten
- Wissen über die Kosten von Operationen

Bestandteile des Optimierers

Der globale Optimierer hat drei wesentliche Bestandteile

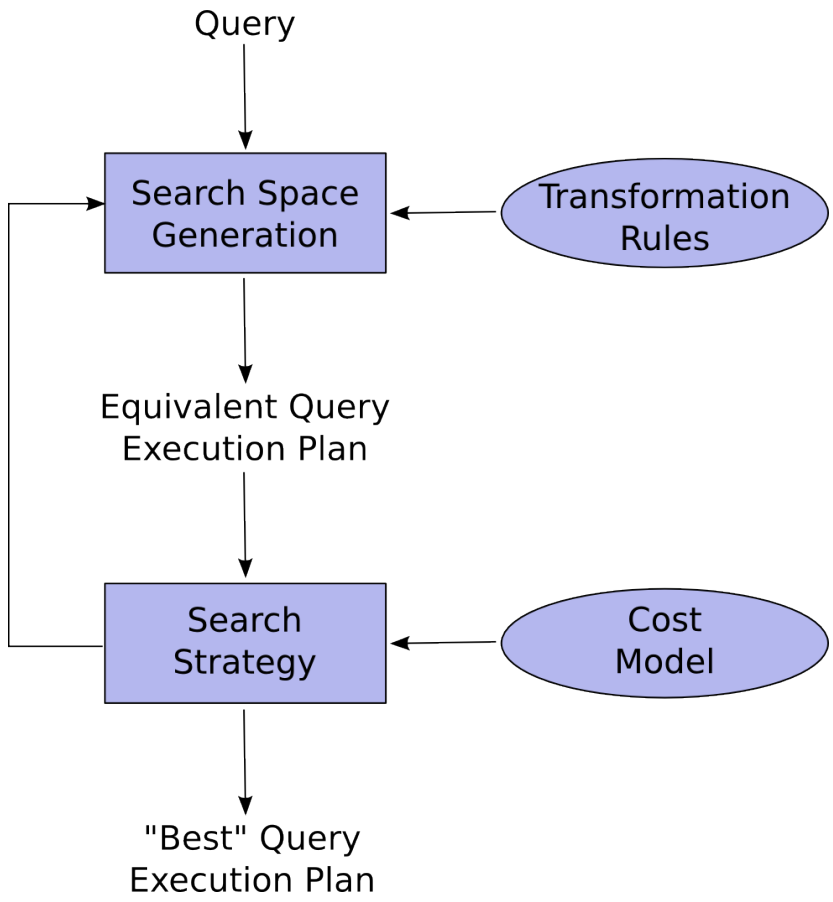
- Der Suchraum
  Menge von äquivalenten alternativen Ausführungsplänen, um die ursprüngliche Anfrage darzustellen
- Das Kostenmodell
  Schätzt die Kosten eines gegebenen Ausführungsplans
- Die Suchstrategie
  Exploriert den Suchraum und wählt den besten Plan

Phasen der Optimierung

Phasen

1. Aufspannen des Suchraums mit Transformationsregeln
   $\rightarrow$ äquivalente Ausführungspläne
2. Anwenden einer Suchstrategie und eines Kostenmodells
   $\rightarrow$ Auswahl eines effizienten Plans

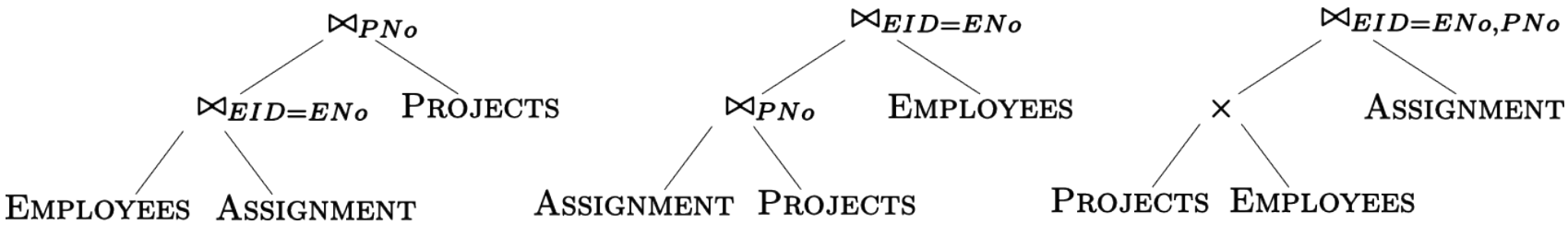
Schwerpunkt: Joinreihenfolge und Joinschachtelung



Anfrage

SELECT EName, Title
FROM Employees e, Assignment a, Project p
WHERE e.EID = ENo **AND** a.PNo=p.PNo

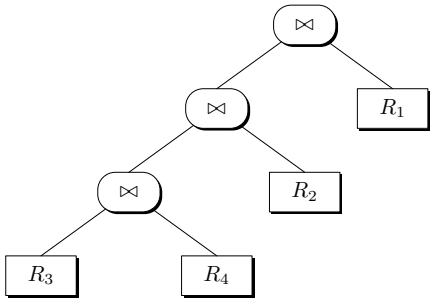
Äquivalente Joinbäume



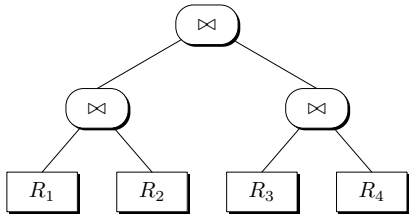
$O(N!)$ verschiedene Joinbäume durch Anwendung von Kommutativitäts- und Assoziativitätsregeln für N Relationen

Baumvarianten für die Optimierung der Joinreihenfolge

- Lineare Joinbäume
  - Alle inneren Knoten haben mindestens einen Blattknoten (Basisrelation) als Kind
  - Schränkt den Suchraum ein
- Buschige Bäume
  - Können innere Knoten haben, die kein Blatt als Kind haben
  - Hohes Potential für Parallelisierung



linear join tree



bushy join tree

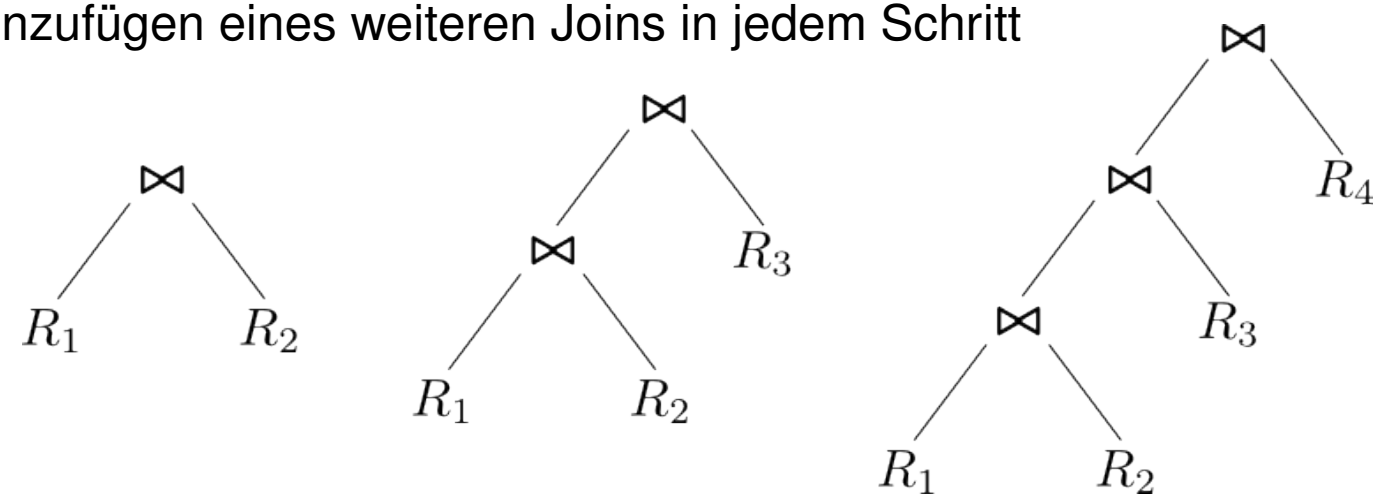
Suchstrategien

Eine Suchstrategie muss den Suchraum verkleinern

- Anwendung von Heuristiken (ähnlich zentraler algebraischer Anfrageoptimierung)
  - Ausführung von Projektionen und Selektionen beim Zugriff auf die Basisrelationen
  - Vermeide Kartesische Produkte – erzwinge Joins
- Anwendung weiterer Heuristiken, die die Gestalt des Joinbaums beeinflussen
  - Reduzieren der Größe des Suchraums vs. Ermöglichen von Parallelisierung *Lineare vs. buschige Bäume*

Deterministische Suchstrategie

- Systematische Generierung von Anfrageplänen
- Beginnend mit Plänen, die auf die Basisrelationen zugreifen
- Aufbau komplexer Pläne durch Kombination einfacher Pläne, z.B. Hinzufügen eines weiteren Joins in jedem Schritt



Erschöpfende Suche garantiert, dass der beste Plan gefunden wird

Suchstrategien

Beispiele für deterministische Suchstrategien

- Dynamische Programmierung
  - (fast) erschöpfende Suche, indem alle möglichen Pläne erstellt werden
  - “Sehr schlechte” Teilpläne werden früh eliminiert
  - Findet garantiert den besten Plan
  - Nur möglich für eine kleine Zahl (5-6) von Relationen
- Greedy-Algorithmus
  - Baut nur einen Plan auf (depth-first)

Suchstrategien

Randomisierte Suchstrategie

- Ein oder wenige Startpläne mittels einer Greedy-Strategie
- Verbesserung der Startpläne durch Untersuchung von “Nachbarplänen”
- Nachbarplan: Anwendung von Transformationsregeln, z.B. Austausch zweier beliebiger Operationen
- Bessere Performance für eine große Zahl von Relationen

Keine Garantie, dass der beste Plan gefunden wird

Verteiltes Kostenmodell

Verteiltes Kostenmodell

Bestandteile

- Kostenfunktion
  Schätzen der Kosten, um Operationen auszuführen
- Statistiken
  Daten über Relationsgrößen, Domänen von Attributen, Werteverteilungen, etc.
- Formeln
  Bestimmen von Kardinalitäten, Größen von Zwischenergebnissen, etc.

Gesamtausführungszeit

- Summe aller Kosten, d.h. Summe aller Verarbeitungszeiten von allen an der Ausführung der Anfrage beteiligten Rechnern

$$T_{\text{total}} = T_{\text{CPU}} \cdot \#insts + T_{\text{I/O}} \cdot \#ops_{I/O} + T_{\text{MSG}} \cdot \#msgs + T_{\text{TR}} \cdot \#bytes$$

  - T_{CPU} Zeit um eine CPU-Anweisung auszuführen
  - $T_{\text{I/O}}$ Zeit für einen Plattenzugriff
  - T_{MSG} Zeit um eine Nachricht zu senden oder zu empfangen
  - T_{TR} Zeit um eine Dateneinheit zwischen zwei Rechnern zu übertragen
    $\#bytes$ ist die Summe der Größen aller Nachrichten
    Typische Annahme: T_{TR} ist konstant – obwohl das für entfernte Rechner nicht stimmen muss

Kostenfunktionen

Bestandteile der Gesamtausführungszeit

- Lokale Verarbeitungskosten bzw. -zeit

$$T_{local} = T_{\text{CPU}} \cdot \#insts + T_{\text{I/O}} \cdot \#ops_{I/O}$$

- Kommunikationskosten bzw. -zeit

$$T_{comm} = T_{\text{MSG}} \cdot \#msgs + T_{\text{TR}} \cdot \#bytes$$

- Die Koeffizienten (T_{CPU}, $T_{\text{I/O}}$, T_{MSG}, T_{TR}) charakterisieren ein bestimmtes verteiltes Datenbanksystem
- WAN (Wide Area Network): Kommunikationszeit dominiert
- LAN (Local Area Network): hier spielen auch die lokalen Kosten eine Rolle

Antwortzeit

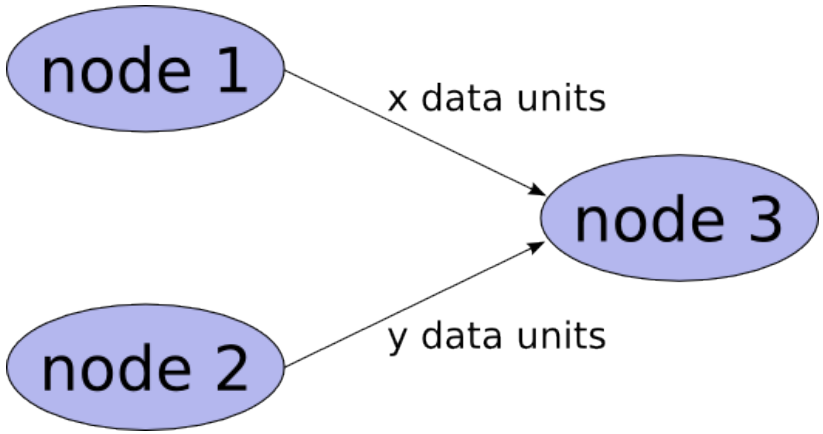
- Zeit zwischen dem Abschicken der Anfrage und ihrer Beendigung
- Berücksichtigung paralleler lokaler Ausführung und paralleler Kommunikation

$$T_{\text{response}} = T_{\text{CPU}} \cdot seq\_\#insts + T_{\text{I/O}} \cdot seq\_\#ops_{I/O} + T_{\text{MSG}} \cdot seq\_\#msgs + T_{\text{TR}} \cdot seq\_\#bytes$$

wobei $seq\_\#x$ die maximale Anzahl von Instruktionen (*insts*), I/O Operationen ($ops_{I/O}$), Nachrichten (*msgs*), oder Bytes (*bytes*) repräsentiert, die sequentiell verarbeitet werden müssen

Gesamtausführungszeit vs. Antwortzeit

Kommunikationskosten



$$T_{comm_{\text{total}}} = 2 \cdot T_{\text{MSG}} + T_{\text{TR}} \cdot (x + y)$$

$$T_{comm_{\text{response}}} = \max\{T_{\text{MSG}} + T_{\text{TR}} \cdot x, T_{\text{MSG}} + T_{\text{TR}} \cdot y\}$$

Die Minimierung der Antwortzeit bedeutet *nicht*, dass gleichzeitig die Gesamtausführungszeit minimiert wird!

Statistiken

Gute Statistiken sind enscheidend

- Wichtigster Kostenfaktor:
  Größe von Zwischenergebnissen, die während der Ausführung erzeugt werden
- Schätzen der Größe mit Statistiken und Formeln
- Tradeoff zwischen Genauigkeit und Kosten zur Verwaltung der Statistiken

Typische Statistiken

Typische Statistiken für Relation R, die in $R_1, R_2, \ldots, R_r$ fragmentiert wurde und Attribute $A_1, \ldots, A_n$ hat

- Länge jedes Attributs A_i in Bytes: $length(A_i)$
- Anzahl verschiedener Werte für jedes Attribut A_i und für jedes Fragment R_j: $values_{A_i,R_j} := card(\pi_{A_i}(R_j))$
- Minimale und maximale Attributwerte: $min(A_i)$ and $max(A_i)$
- Anzahl verschiedener Werte (Kardinalität) der Domänen der Attribute: $card(dom[A_i])$
- Anzahl von Tupeln in jedem Fragment R_j: $card(R_j)$

Zusätzliche Statistiken

- Histogramm für jedes Attribut A_i, um die Verteilung der Wertehäufigkeiten zu approximieren
- Joinselektivitätsfaktoren für einige Paare von Relationen

$$SF_J(R,S) = \frac{card(R \bowtie S)}{card(R) \cdot card(S)}$$

gute (hohe) Selektivität: $SF_J = 0.001$
schlechte (niedrige) Selektivität: $SF_J = 0.5$

Annahmen

- Attribute sind voneinander unabhängig
- Werte der Attribute sind gleichverteilt

Selektivität

- Verhältnis zwischen erwarteter Anzahl von Ergebnistupeln und Anzahl der Tupel in der Eingaberelation

$$SF = \frac{\text{Erwartete Ergebnisgröße}}{\text{Größe der Eingaberelation}}$$

- Beispiel: $\sigma_F(R)$ gibt 10% von R's Tupeln zurück $\rightsquigarrow SF_S(F, R) = 0.1$ (*SF Selektivitätsfaktor*)

Annahmen

- Attribute sind voneinander unabhängig
- Werte der Attribute sind gleichverteilt

Kardinalität

- Schätze die Ergebnisgröße (Kardinalität der Ausgaberelation)
- Beispiel: $SF_S(F, R) = 0.1$

$$card(\sigma_F(R)) = SF_S(F, R) \cdot card(R)$$

Kardinalität

$$card(\sigma_F(R)) = SF_S(F, R) \cdot card(R)$$

Selektivität

- Die Selektivität hängt ab von den Selektionsprädikaten $p(A)$ und Konstanten v

$$SF_S(A = v, R) = \frac{1}{values_{A,R}} = \frac{1}{card(\pi_A(R))}$$

$$SF_S(A < v, R) = \frac{v - min(A)}{max(A) - min(A)}$$

$$SF_S(A > v, R) = \frac{max(A) - v}{max(A) - min(A)}$$

$$SF_S(v_1 < A < v_2, R) = \frac{v_2 - v_1}{max(A) - min(A)}$$

Kardinalität

$$card(\sigma_F(R)) = SF_S(F, R) \cdot card(R)$$

Selektivität

- Die Selektivität hängt ab von den Selektionsprädikaten $p(A)$ und Konstanten v

$$\begin{aligned} SF_S(p(A_i) \wedge p(A_j), R) &= SF_S(p(A_i), R) \cdot SF_S(p(A_j), R) \\ SF_S(p(A_i) \vee p(A_j), R) &= SF_S(p(A_i), R) + SF_S(p(A_j), R) - \\ &\quad (SF_S(p(A_i), R) \cdot SF_S(p(A_j), R)) \end{aligned}$$

Kardinalität

- Ohne Duplikateliminierung

$$card(\pi_A(R)) = card(R)$$

- Mit Duplikateliminierung (für ein beliebiges Attribut A):

$$card(\pi_A(R)) = values_{A,R}$$

- Mit Duplikateliminierung (wenn eines der Attribute ein Primärschlüssel ist):

$$card(\pi_{A_i,\ldots}(R)) = card(R)$$

Kardinalitäten von Projektionen auf beliebige Kombinationen von Attributen sind schwierig zu schätzen, da Attributkorrelationen in der Regel unbekannt sind

Kartesisches Produkt

Kardinalität

$$card(R \times S) = card(R) \cdot card(S)$$

Kartesisches Produkt

- Gegeben: $R \bowtie S$ mit $R(A, B)$ und $S(B, C)$
- Obere Schranke: Größe des kartesischen Produktes

Natürlicher Join auf Attribut B

- Keine Werte von B gemeinsam zwischen R und S:
$$card(R \bowtie S) = 0$$
- Fremdschlüsselbeziehung $R.B \rightarrow S.B$:
$$card(R \bowtie S) = card(R)$$
- Alle Tupel in $R.B$ und $S.B$ haben den gleichen Wert:
$$card(R \bowtie S) = card(R) \cdot card(S)$$

Kardinalität

- Gegeben: $R \bowtie S$ mit $R(A, B)$ und $S(B, C)$
- Obere Schranke: Größe des kartesischen Produktes

Natürlicher Join auf Attribut B

- Schätze

$$card(R \bowtie S) = \frac{card(R) \cdot card(S)}{\max\{values_{B,R},\ values_{B,S}\}}$$

- Speichere Statistiken (Joinkardinalität SF_J) für wichtige Joins

$$card(R \bowtie S) = SF_J \cdot card(R) \cdot card(S)$$

Union und Differenz

Kardinalität

- Schwierig zu schätzen, weil Duplikate entfernt werden
- Union
  - Obere Schranke

$$card(R \cup S) = card(R) + card(S)$$

  - Untere Schranke

$$card(R \cup S) = max\{card(R), card(S)\}$$

- Differenz
  - Obere Schranke

$$card(R \setminus S) = card(R)$$

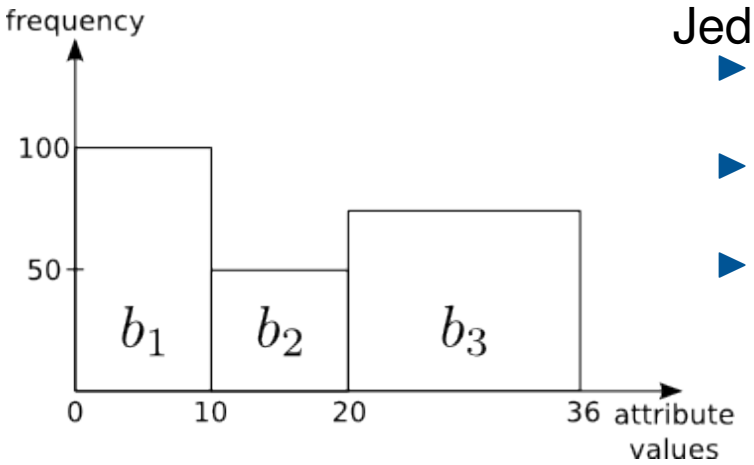
  - Untere Schranke

$$card(R \setminus S) = 0$$

Histogramme

- In der Realität sind die Werte von Attributen oft *nicht uniform* verteilt
- Histogramme bestehen aus einer Menge von Buckets b_i

Beispielhistogramm auf Attribut A von Relation R



Jeder Bucket b_i wird definiert durch

- Bereich: $range_i$
  Bereich der Werte in der Attributdomäne $dom[A]$
- Frequenz: f_i
  Anzahl der Tupel von R für die $R.A \in range_i$
- Unterschiedliche Werte: d_i
  Anzahl der unterschiedlichen Werte von A wobei $R.A \in range_i$

Prädikat mit Gleichheit

- Gegeben Prädikat $A = v$
- Finde Bucket b_i so dass $v \in range_i$

$$SF_S(A = v, R) = \frac{1}{d_i}$$

$$card(\sigma_{A=v}(R)) = SF_S(A = v, R) \cdot f_i = \frac{f_i}{d_i}$$

Selektivitätsschätzung mit Histogrammen

Bereichsprädikate

- Gegeben Prädikat $A \leq v$
- Finde Buckets, die mit dem Anfragebereich überlappen
- Addiere die Frequenzen

$$card(\sigma_{A \leq v}(R)) = \sum_{j=1}^{i-1} f_j + \left(\frac{v - min(range_i)}{max(range_i) - min(range_i)} \cdot f_i \right)$$

Bucket i überlappt nur teilweise mit dem Anfragebereich

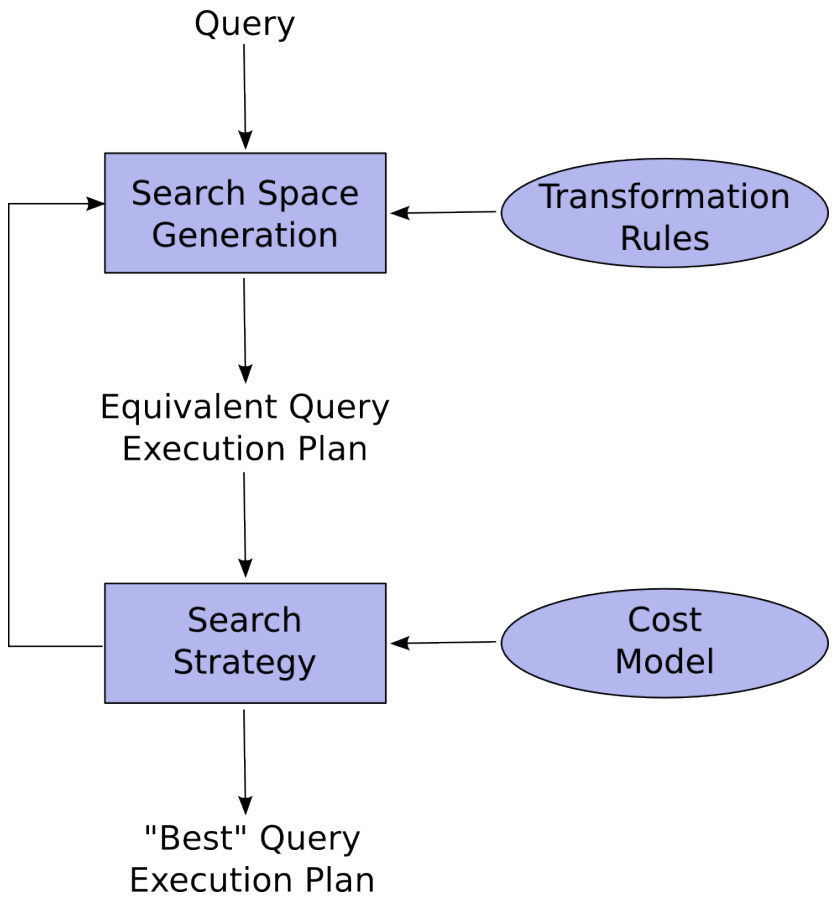
Optimierung der Joinreihenfolge

Phasen der Optimierung

Phasen

1. Aufspannen des Suchraums mit Transformationsregeln
   $\rightarrow$ äquivalente Ausführungspläne
2. Anwenden einer Suchstrategie und eines Kostenmodells
   $\rightarrow$ Auswahl eines effizienten Plans

Schwerpunkt: Joinreihenfolge und Joinschachtelung



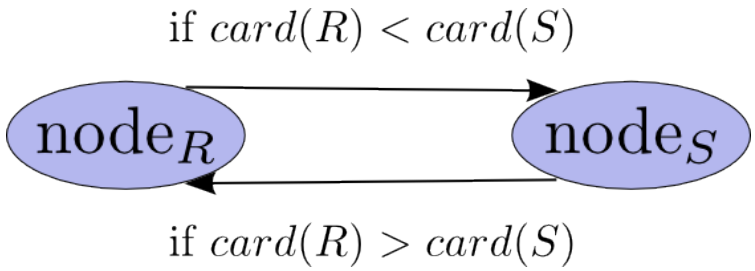
Optimierung der Joinreihenfolge

Vereinfachende Annahmen

- Keine Unterscheidung zwischen Fragmenten und Relationen
- Keine Berücksichtigung der lokalen Verarbeitungszeit
- Keine Berücksichtigung anderer Operationen (Selektion, Projektion)
- Kein Pipelining
- Keine Berücksichtung des Datentransfers zum anfragenden Rechner

Optimierung der Joinreihenfolge für zwei Relationen

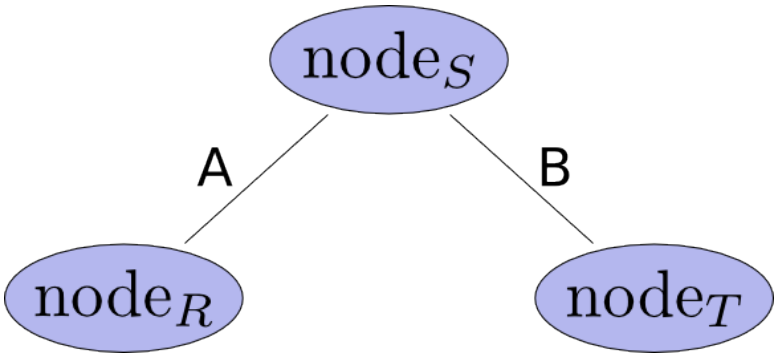
Bestimme die Joinreihenfolge für zwei Relationen $R \bowtie S$



Übertrage die kleinere Relation, um die Netzlast zu minimieren

Optimierung der Joinreihenfolge für drei Relationen

Bestimme die Joinreihenfolge für drei Relationen $R \bowtie_A S \bowtie_B T$



1. $R \rightsquigarrow node_S$, $node_S$: $R' = R \bowtie S$, $R' \rightsquigarrow node_T$, $node_T$: $R' \bowtie T$
2. $S \rightsquigarrow node_R$, $node_R$: $R' = R \bowtie S$, $R' \rightsquigarrow node_T$, $node_T$: $R' \bowtie T$
3. $S \rightsquigarrow node_T$, $node_T$: $S' = S \bowtie T$, $S' \rightsquigarrow node_R$, $node_R$: $S' \bowtie R$
4. $T \rightsquigarrow node_S$, $node_S$: $S' = S \bowtie T$, $S' \rightsquigarrow node_R$, $node_R$: $S' \bowtie R$
5. $T \rightsquigarrow node_S$, $R \rightsquigarrow node_S$, $node_S$: $R \bowtie S \bowtie R$

Mögliche Reihenfolgen

1.
   - $node_R$: sende R an $node_S$
   - $node_S$: berechne Join $R' = R \bowtie S$, sende R' an $node_T$
   - $node_T$: berechne Join $R' \bowtie T$
2.
   - $node_S$: sende S an $node_R$
   - $node_R$: berechne Join $R' = R \bowtie S$, sende R' an $node_T$
   - $node_T$: berechne Join $R' \bowtie T$
3.
   - $node_S$: sende S an $node_T$
   - $node_T$: berechne Join $S' = S \bowtie T$, sende S' an $node_R$
   - $node_R$: berechne Join $S' \bowtie R$
4.
   - $node_T$: sende T an $node_S$
   - $node_S$: berechne Join $S' = S \bowtie T$, sende S' an $node_R$
   - $node_R$: berechne Join $S' \bowtie R$

Optimierung der Joinreihenfolge mit Semijoins

Berücksichtigung von **Semi-Joins**, um zwei Relationen R (auf Rechner $node_R$) und S (auf Rechner $node_S$) zu joinen, ergibt drei Alternativen – unter der Annahme, dass A das Joinattribut ist

1. $R \bowtie_A S = (R \ltimes_A S) \bowtie_A S = (R \bowtie_A \pi_A(S)) \bowtie_A S$
2. $R \bowtie_A S = R \bowtie_A (S \ltimes_A R)$
3. $R \bowtie_A S = (R \ltimes_A S) \bowtie_A (S \ltimes_A R)$

Workflow für Alternative 1

- $node_S$: berechne $S' = \pi_A(S)$, sende S' an $node_R$
- $node_R$: berechne $R' = R \ltimes_A S'$, sende R' an $node_S$
- $node_S$: berechne $R' \bowtie_A S$

Transferkosten (ohne Berücksichtigung von T_{MSG})

- $T_{TR} \cdot card(\pi_A(S)) + T_{TR} \cdot card(R \ltimes_A S')$
- Wenn nur volle Joins ($R \bowtie_A S$) berücksichtigt werden und wir annehmen, dass $card(R) < card(S)$, würde die komplette Relation R an $node_S$ geschickt mit Kosten $T_{TR} \cdot card(R)$

Semijoin vs. Join

Folgerung

- Transferkosten mit Semijoin: $T_{TR} \cdot card(\pi_A(S)) + T_{TR} \cdot card(R \ltimes_A S)$
- Transferkosten mit Standardjoin: $T_{TR} \cdot card(R)$

Der Semijoin ist zu bevorzugen, wenn

$$card(\pi_A(S)) + card(R \ltimes_A S) < card(R)$$

Modelle für die Gesamtausführungszeit

Modelle für die Gesamtausführungszeit

Grundlegende Strategie

- Es gibt einen Koordinator (Master)
- Erschöpfende Suche
- Optimierungsziel: Gesamtausführungszeit

Ausführungszeit

- Relationaler Operatorbaum
- Kostenmodell
- Statistiken
- Speicherort der Relationen

Ausgabe

- Optimierter Ausführungsplan

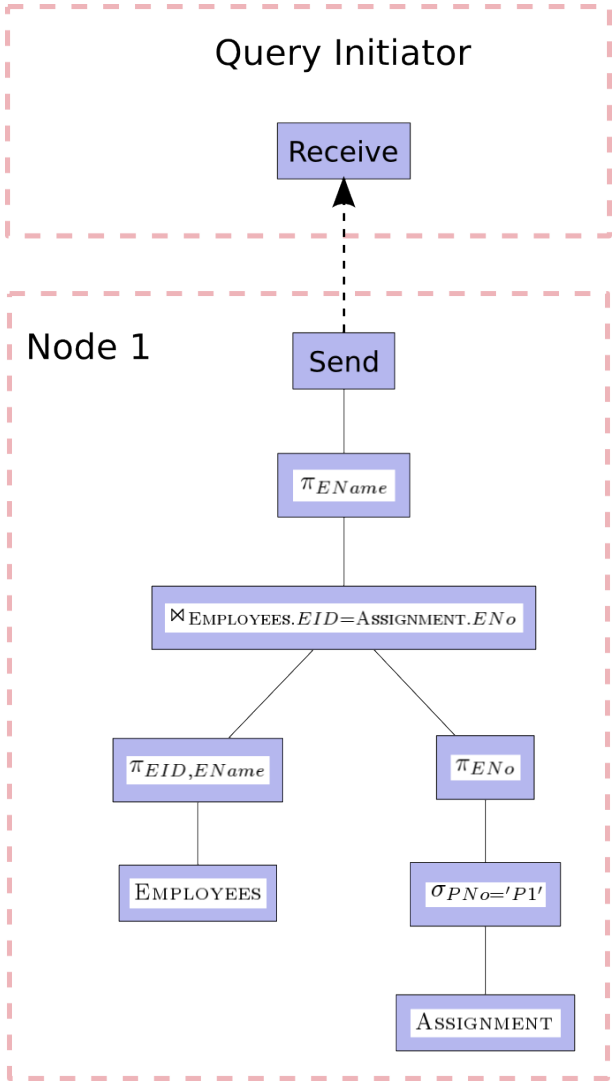
Modelle für die Gesamtausführungszeit

Aspekte

- Kostenmodell
- Rechnerauswahl und Datentransfer
- Optimierung der Joinreihenfolge
- Implementierung der Joins

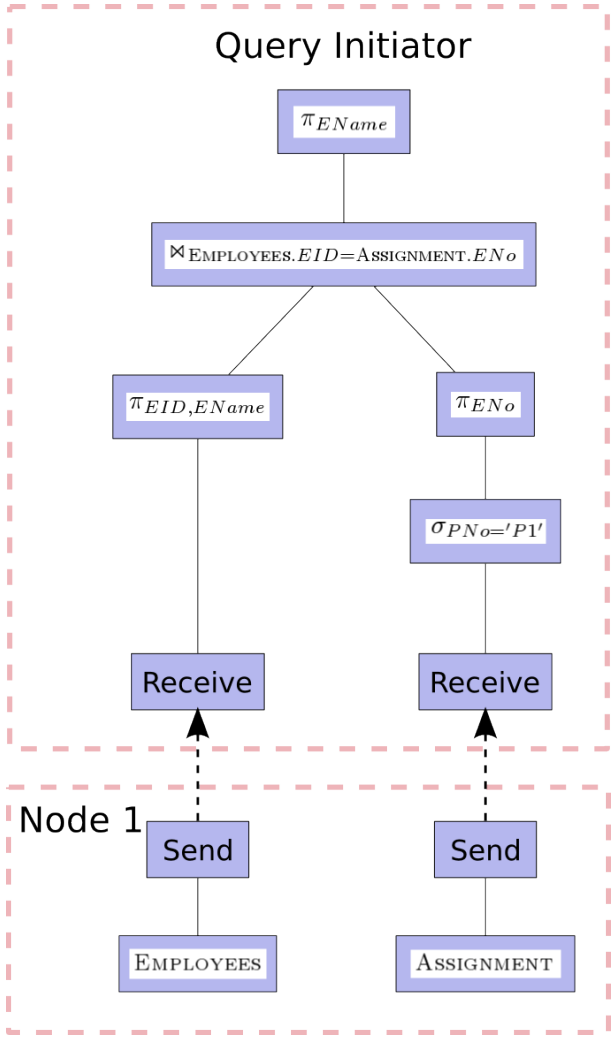
Versenden der Anfrage (Query Shipping)

- Der Initiator der Anfrage (der Rechner, der die Anfrage abschickt bzw. optimiert hat) sendet die Anfrage an andere Rechner
- Empfängerrechner berechnen die Ergebnisse der Anfrage und schicken das Ergebnis zurück zum Initiator



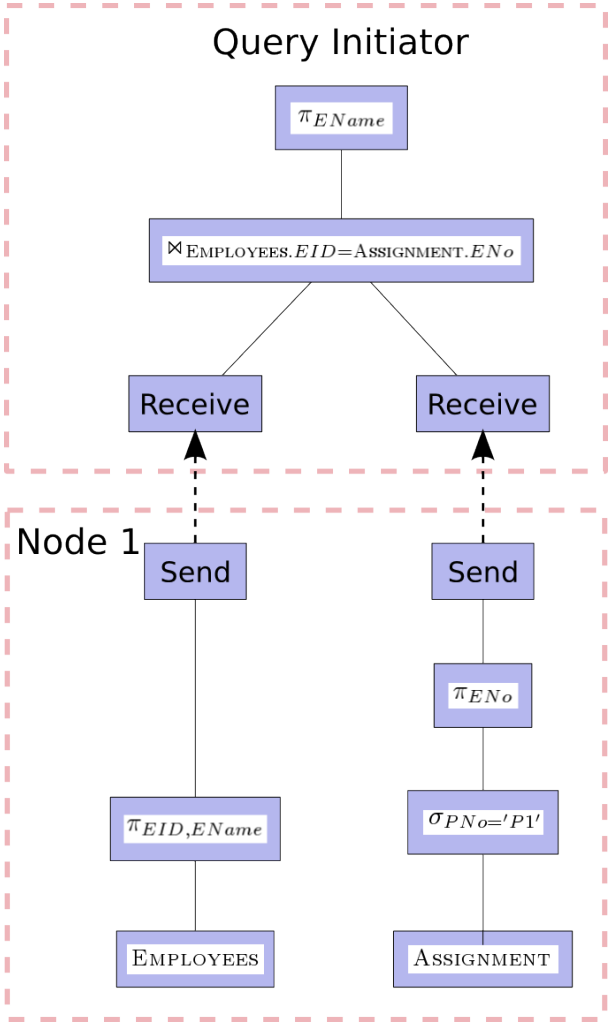
Versenden der Daten (Data Shipping)

- Die Anfrage bleibt beim Initiator
- Der Initiator fordert die benötigten Daten bei anderen Rechnern an
- Empfängerrechner senden alle benötigten Daten an den Initiator
- Der Initiator berechnet das Ergebnis



Hybrides Versenden

- Der Initiator sendet Teilanfragen an andere Rechner
- Andere Rechner führen Teilanfragen aus und schicken Zwischenergebnisse an den Initiator
- Der Initiator führt die verbleibenden Operationen aus (Post-Processing)



Rechnerauswahl und Datentransfer für Joins

Problem

- Anfragen verwenden viele Joins
- Die Berechnung von Joins kann sehr teuer sein
- vor allem in verteilten System: besondere Aufmerksamkeit wegen Fragmenten und Replikation notwendig

Grundlegende Strategien

- komplett versenden (ship whole)
  Übertragen der kompletten Relation
- nach Bedarf übertragen (fetch as needed)
  Stückweises Übertragen der Relation

Rechnerauswahl und Datentransfer für Joins

Szenario

- 2 Rechner; einer ($node_R$) speichert Relation R, der andere ($node_S$) Relation S
- Die Anfrage besteht aus $R \bowtie S$

R

A	B
3	7
1	1
4	6
7	7
4	5
6	2
5	7

S

B	C	D
9	8	8
1	5	1
9	4	2
4	3	3
4	2	6
5	7	8

$R \bowtie S$

A	B	C	D
1	1	5	1
4	5	7	8

Ship whole

R

A	B
3	7
1	1
4	6
7	7
4	5
6	2
5	7

S

B	C	D
9	8	8
1	5	1
9	4	2
4	3	3
4	2	6
5	7	8

$R \bowtie S$

A	B	C	D
1	1	5	1
4	5	7	8

Ausführung auf Rechner $node_R$

- $node_R$: verlangt Übertragung von Relation S von $node_S$
- $node_S$: schickt die gewünschten Daten (Relation S) an $node_R$

Gesamtkosten: 2 Nachrichten, 18 Attributwerte

Ship whole

R

A	B
3	7
1	1
4	6
7	7
4	5
6	2
5	7

S

B	C	D
9	8	8
1	5	1
9	4	2
4	3	3
4	2	6
5	7	8

$R \bowtie S$

A	B	C	D
1	1	5	1
4	5	7	8

Ausführung auf Rechner $node_S$

- $node_S$: verlangt Übertragung von Relation R von $node_R$
- $node_R$: schickt die verlangten Daten (Relation R) an $node_S$

Gesamtkosten: 2 Nachrichten, 14 Attributwerte

Ship whole

R

A	B
3	7
1	1
4	6
7	7
4	5
6	2
5	7

S

B	C	D
9	8	8
1	5	1
9	4	2
4	3	3
4	2	6
5	7	8

R $\bowtie$ S

A	B	C	D
1	1	5	1
4	5	7	8

Ausführung auf einem dritten Rechner $node_X$

- $node_X$: verlangt Übertragung von Relation R von $node_R$
- $node_X$: verlangt Übertragung von Relation S von $node_S$
- $node_R$: schickt die verlangten Daten (Relation R) an $node_X$
- $node_S$: schickt die verlangten Daten (Relation S) an $node_X$

Gesamtkosten: 4 Nachrichten, $18 + 14 = 32$ Attributwerte

Fetch as needed

R

A	B
3	7
1	1
4	6
7	7
4	5
6	2
5	7

S

B	C	D
9	8	8
1	5	1
9	4	2
4	3	3
4	2	6
5	7	8

R $\bowtie$ S

A	B	C	D
1	1	5	1
4	5	7	8

Ausführung auf Rechner $node_R$

- $node_R$: verlangt Übertragung der Tupel von Relation S mit $B = 7$ von $node_S$
- $node_S$: sendet gewünschte Tupel (0 Tupel von Relation S mit $B = 7$) an $node_R$
- $node_R$: verlangt Übertragung der Tupel von Relation S mit $B = 1$ von $node_S$
- $node_S$: sendet gewünschte Tupel (1 Tupel von Relation S mit $B = 1$) an $node_R$
- ...

Gesamtkosten: $7 \cdot 2 = 14$ Nachrichten, $7 + 2 \cdot 3 = 13$ Attributwerte

Fetch as needed

R

A	B
3	7
1	1
4	6
7	7
4	5
6	2
5	7

S

B	C	D
9	8	8
1	5	1
9	4	2
4	3	3
4	2	6
5	7	8

R $\bowtie$ S

A	B	C	D
1	1	5	1
4	5	7	8

Ausführung auf $node_S$

- $node_S$: verlangt Übertragung der Tupel von Relation R mit $B = 9$ von $node_R$
- $node_R$: sendet gewünschte Tupel (0 Tupel von Relation R mit $B = 9$) an $node_S$
- $node_S$: verlangt Übertragung der Tupel von Relation R mit $B = 1$ von $node_R$
- $node_R$: sendet gewünschte Tupel (1 Tupel von Relation R mit $B = 1$) an $node_S$
- ...

Gesamtkosten: $6 \cdot 2 = 12$ Nachrichten, $6 + 2 \cdot 2 = 10$ Attributwerte

Ship whole vs. fetch as needed

Folgerungen

- Fetch as needed generiert eine höhere Anzahl von Nachrichten
- Ship whole generiert mehr Datentransfer

Fortgeschrittene Strategien, die auf diesen beiden einfachen Strategien aufbauen

- Nested loops join
- Sort-merge
- Semijoin
- Bitvektor-Join

Nested loops join

Doppelte Schleife, die über alle $t_r \in R$ und alle $t_s \in S$ iteriert, um $R \bowtie_F S$ zu berechnen

Algorithmus

for each $t_r \in R$ **do**
    for each $t_s \in S$ **do**
        if t_r and t_s fulfill the join predicate F
            then add their concatenation to the result

Merge join

Erfordert, dass beide Joinrelationen (R und S) nach den Attributen sortiert sind, die im Joinprädikatvorkommen; normalerweise nur bei einfachen Equi-Joins verwendet

Algorithmus

t_r = first tuple in R
t_s = first tuple in S
  while $t_r \neq endof(R)$ **and** $t_s \neq endof(S)$ **do**
    if $t_r < t_s$
      t_r = next $t_r \in R$
    if $t_r > t_s$
      t_s = next $t_s \in S$
    if $t_r = t_s$
      concatenate t_r with t_s and all following entries in S that have
        the same values for the join attributes as t_s
      do the same for all $t_r \in R$ with the same join attribute values as t_r
      set t_r and t_s to first entries that are unequal to the previously concatenated t_r and t_s va

Semijoin

Anforderung aller Joinpartner in einem Schritt

Grundlegende Überlegung:

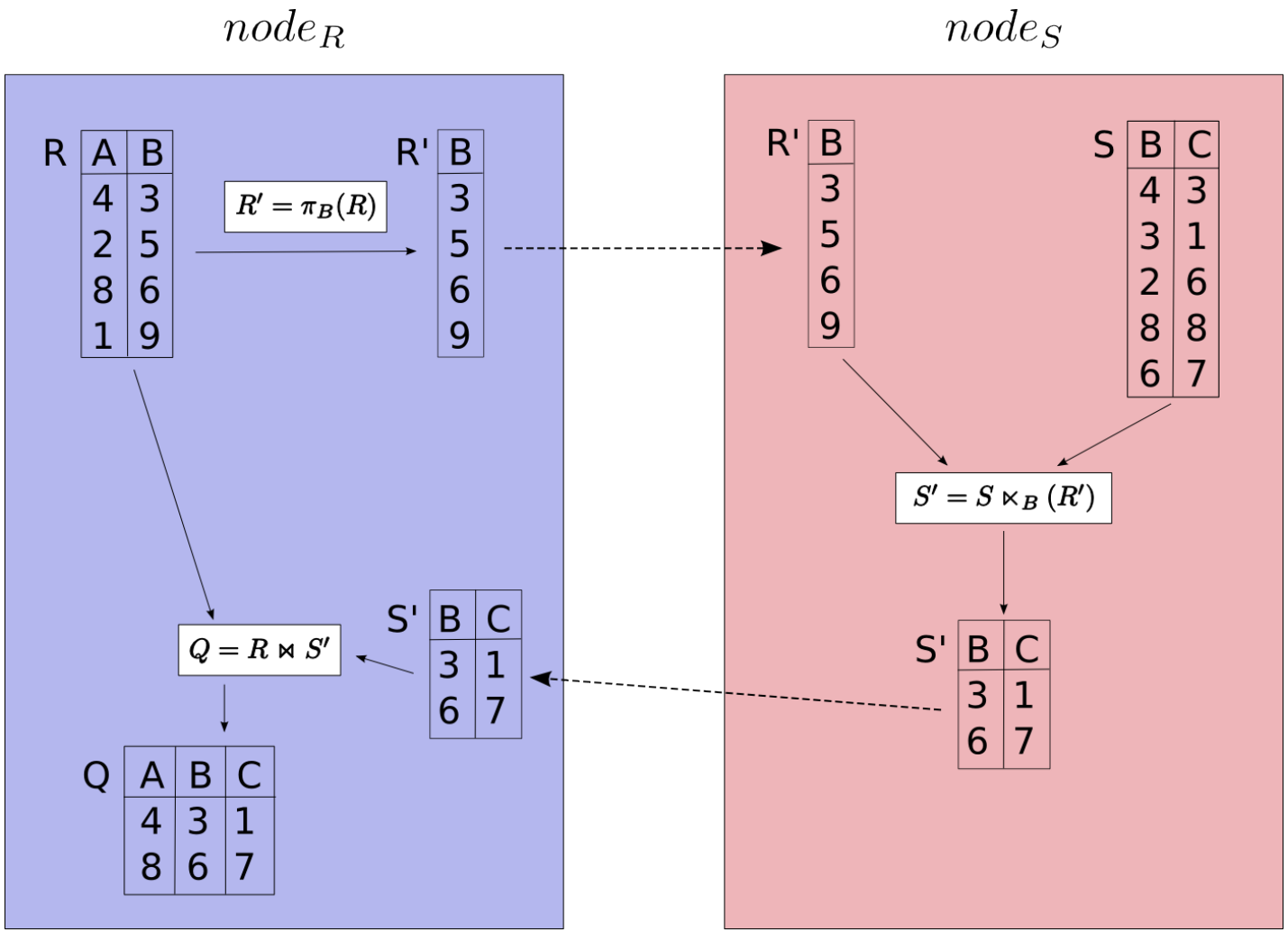
$$R \bowtie S = R \bowtie (S \ltimes R) = R \bowtie (S \bowtie \pi_B(R))$$

wobei B das Joinattribut ist

Algorithmus

- $node_R$: bestimme $\pi_B(R)$ und schicke das Ergebnis an $node_S$
- $node_S$: bestimme $S' = S \bowtie \pi_B(R) = S \ltimes R$ und schicke das Ergebnis an $node_R$
- $node_R$: bestimme $R \bowtie S' = R \bowtie S$

Semijoin


$node_R$
$node_S$
R
A B
4 3
2 5
8 6
1 9
$R' = \pi_B(R)$
R'
B
3
5
6
9
R'
B
3
5
6
9
S
B C
4 3
3 1
2 6
8 8
6 7
$S' = S \ltimes_B (R')$
S'
B C
3 1
6 7
S'
B C
3 1
6 7
$Q = R \ltimes S'$
Q
A B C
4 3 1
8 6 7

Bitvektor-Join

- Auch bekannt als Hashfilter-Join
- Vermeide es, alle Werte der Joinattribute zum anderen Rechner zu übertragen
- Übertrage stattdessen Bitvektor $BV[1 \ldots n]$

Transformation

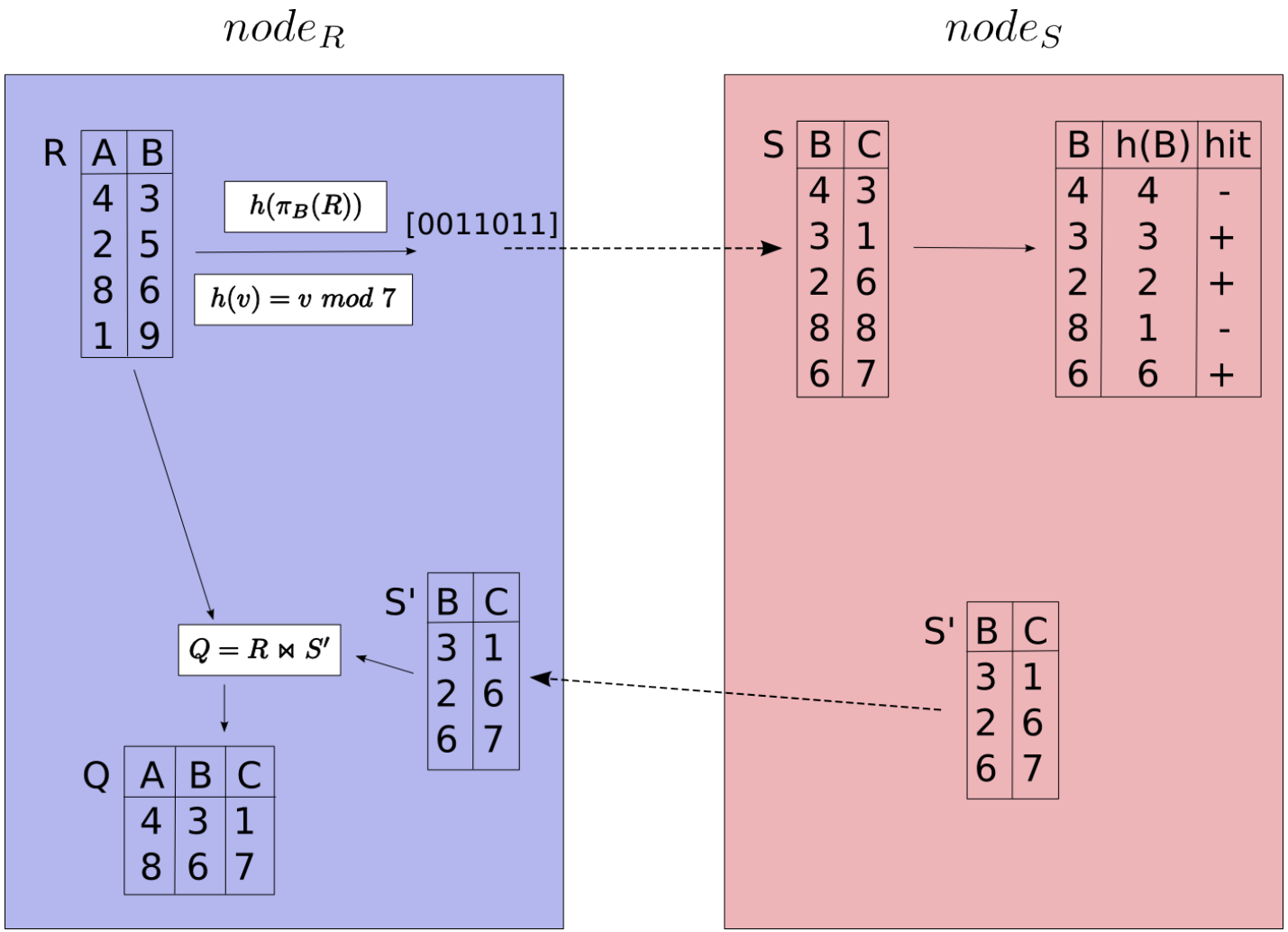
- Wähle eine geeignete Hashfunktion h
- Wende h an, um die Attributwerde in den Bereich $[1 \ldots n]$ zu transformieren
- Setze die entsprechenden Bits im Bitvektor $BV[1 \ldots n]$ auf 1

Bitvektor-Join

Algorithmus

- $node_R$: bestimme $\pi_B(R)$, wende die Hashfunktion h auf das Ergebnis an, setze die entsprechenden Bits in BV auf 1, und sende das Ergebnis an $node_S$
- $node_S$: wende die Hashfunktion h auf das Joinattribut von Relation S an, bestimme $S' = \{t \in S | BV[h(t.B)] = 1\}$, sende S' an $node_R$
- $node_R$: bestimme $R \bowtie S' = R \bowtie S$

Bitvektor-Join



Bitvektor-Join

Zusammenfassung

- Übertragung des Bitvektors reduziert die Netzlast
- Der Bitvektor gibt nur einen Hinweis auf potentielle Joinpartner, da mehrere Attributwerte auf den gleichen Hashwert abgebildet werden können
  Kann dazu führen, dass unnötige Tupel übertragen werden
- Anforderungen: Eine geeignete Hashfunktion h, und n muss so groß sein, dass keine große Anzahl von Hashkollisionen entsteht

Antwortzeitmodelle

Antwortzeitmodelle

- “klassische” Kostenmodelle betrachten den gesamten Resourcenverbrauch einer Anfrage
  - Gute Ergebnisse für hohe Rechenlast und langsame Netzverbindungen
    Wenn Resourcen gespart werden, können viele Anfragen parallel verarbeitet werden (minimale Last, maximaler Durchsatz)
- Optimierung für kurze Antwortzeiten
  - “verschwende” Resourcen, um Ergebnisse der Anfrage früher zu erhalten
  - Nutze schwach ausgelastete Rechner und schnelle Verbindungen aus
  - Nutze Parallelität innerhalb der Anfrage aus

Antwortzeitmodelle

Zwei verschiedene Antworteiten

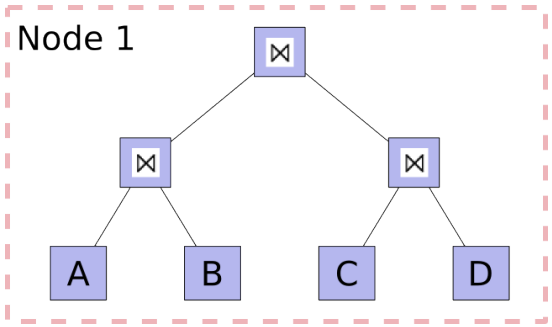
- Wann trifft das erste Ergebnistupel ein?
- Wann sind alle Ergebnistupel eingetroffen?

Beispielsituation

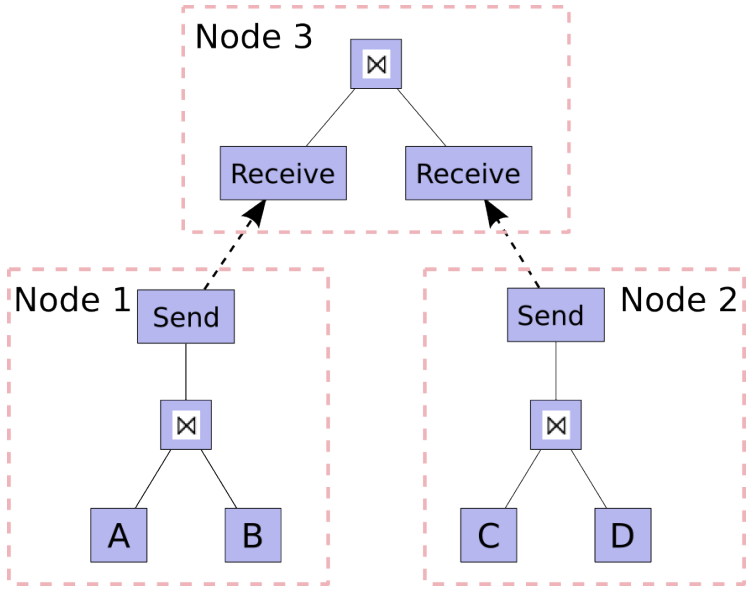
- Gegeben Relationen/Fragmente A, B, C und D
- Volle Replikation, d.h. alle Relationen/Fragmente sind auf allen Rechnern verfügbar
- Berechne $(A \bowtie B) \bowtie (C \bowtie D)$
- Annahmen
  - Jeder Join kostet 20 Zeiteinheiten ($T_{CPU} + T_{I/O}$)
  - Übertragen eines Zwischenergebnisses kostet 10 Zeiteinheiten ($T_{MSG} + T_{TR}$)
  - Zugreifen einer Relation ist kostenlos
  - Jeder Rechner hat einen Berechnungsthread

Zwei Pläne

- Plan 1: Führe alle Operationen auf einem Rechner aus
  Gesamtkosten: 60
- Plan 2: Führe die Joins auf verschiedenen Rechnern aus, verschicke die Ergebnisse
  Gesamtkosten: 80



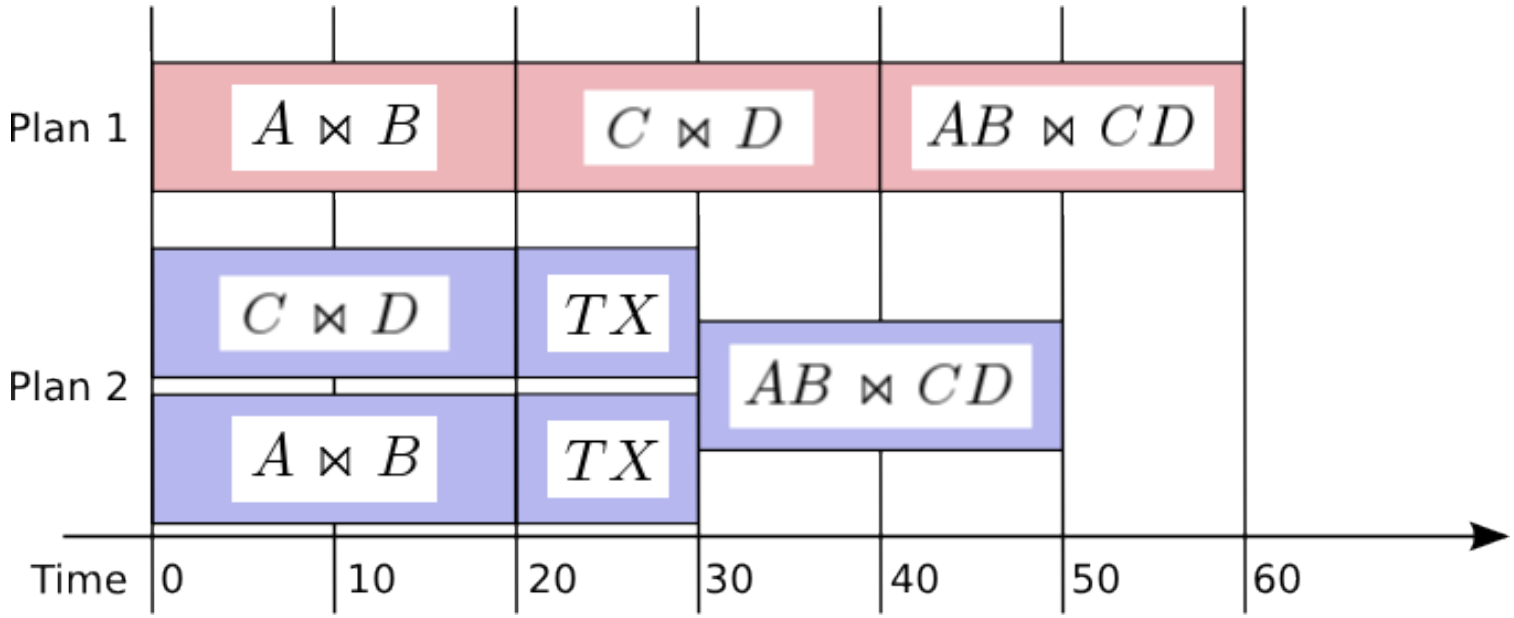
Plan 1



Plan 2

Plan 1 ist offenbar besser im Hinblick auf die Gesamtkosten

Beispiel



Antwortzeit-basierte Kosten: 60 für Plan 1, 50 für Plan 2
$\Rightarrow$ Plan 2 ist besser im Hinblick auf Antwortzeit
Weil Operationen parallel ausgeführt werden können (Ausnutzen von Parallelität innerhalb der Anfrage)
Die Antwortzeit kann weiter verbessert werden durch die Verwendung von Pipelining

Ziel der Verwendung von Pipelining

Gute Antwortzeit für das erste Tupel, indem Anfragen in der Art einer Pipeline ausgeführt werden

- ohne Pipelining
  - Jede Operation wird vollständig abgeschlossen und ein Zwischenergebnis wird materialisiert
  - Die nächste Operation liest das Zwischenergebnis und wird wieder vollständig abgeschlossen
  - Lesen und Schreiben der Zwischenergebnisse bindet Resourcen
- mit Pipelining
  - Operationen generieren keine Zwischenergebnisse
  - Jedes Ergebnistupel wird sofort an die nächste Operation gegeben
  - Tupel “fließen” durch die Operationen

Probleme

- Operationen haben verschiedene Bearbeitungszeiten
  Wenn sich die Ausführungsgeschwindigkeit von Operationen in der Pipeline unterscheidet, werden Tupel gecacht oder die Pipeline wird blockiert
- Manche Operationen sind besser geeignet als andere
  - gut: scan, select, project, union, ...
  - schwierig: join, intersection, ...
  - sehr schwierig: Sortieren

Beispiel für Pipelining

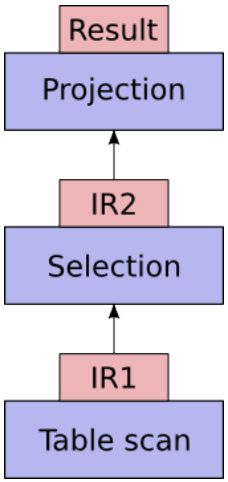
Einfache Anfrage in einem Thread

- Tablescan, selection, projection
  1000 Tupel werden gescannt, Selektivität ist 0.1

Kosten

- Zugriff auf ein Tupel während des Tablescans: 2 Zeiteinheiten
- Selektion (Testen) eines Tupels: 1 Zeiteinheit
- Projizieren eines Tupels: 1 Zeiteinheit

ohne Pipelining



time	Ereignis
2	erstes Tupel in IR1
2000	alle Tupel in IR1
2001	erstes Tupel in IR2
3000	alle Tupel in IR2
3001	erstes Tupel in Result
3100	alle Tupel in Result

Beispiel für Pipelining

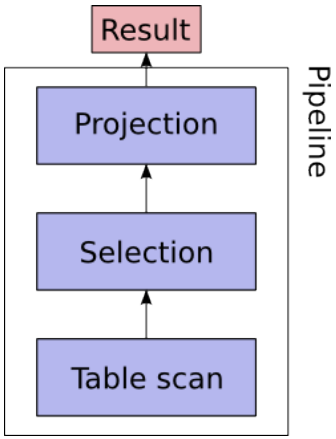
Einfache Anfrage in einem Thread

- Tablescan, selection, projection
  1000 Tupel werden gescannt, Selektivität ist 0.1

Kosten

- Zugriff auf ein Tupel während des Tablescans: 2 Zeiteinheiten
- Selektion (Testen) eines Tupels: 1 Zeiteinheit
- Projizieren eines Tupels: 1 Zeiteinheit

mit Pipelining

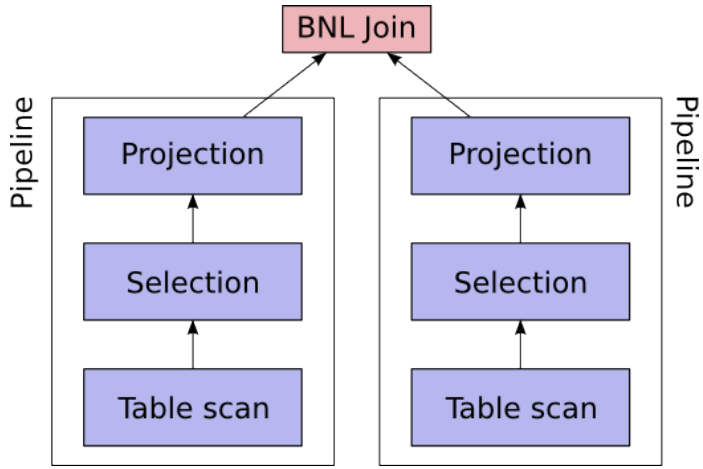


time	Ereignis
2	erstes Tupel beendet Tablescan
3	erstes Tupel beendet Selektion (wenn ausgewählt. . .)
4	erstes Tupel in Result
3098	letztes Tupel beendet Tablescan
3099	letztes Tupel beendet Selektion
3100	alle Tupel in Result

Beispiel für Pipelining

Anfrage mit Join

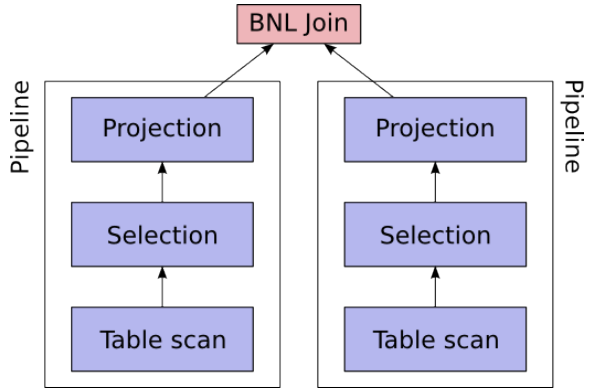
- Joinen von Teilmengen zweier Tabellen mit einem nicht-gepipelineten BNL (block-nested-loop) Join
- Beide Pipelines arbeiten parallel



Beispiel für Pipelining

Kosten

- 1000 werden in jeder Pipeline gescannt, Selektivität ist 0.1
- Joinen von 100 $\bowtie$ 100 Tupeln: 10.000 Zeiteinheiten (eine Zeiteinheit je Kombination)



Antwortzeit

- Das erste Tupel erscheint am Ende einer beliebigen Pipeline nach 4 Zeiteinheiten
- Alle Tupel sind am Ende der Pipelines angekommen nach 3.100 Zeiteinheiten
- Das finale Ergebnis ist verfügbar nach 13.100 Zeiteinheiten
  - Kein Nutzen aus dem Pipelining, was die Antwortzeit angeht
  - Das erste Tupel der Antwort erscheint lange nach Schritt 3.100

Joins und Pipelining

Suboptimales Ergebnis wegen des Joins, der kein Pipelining unterstützt

- Die meisten traditionellen Joinverfahren sind für Pipelining nicht geeignet
- Single-/semi-pipelined: Nur eine Pipeline, das andere Zwischenergebnis muss materialisiert werden
- Voll gepipelined: beide Eingaben werden in der Art einer Pipeline verarbeitet

Single-pipelined hash join

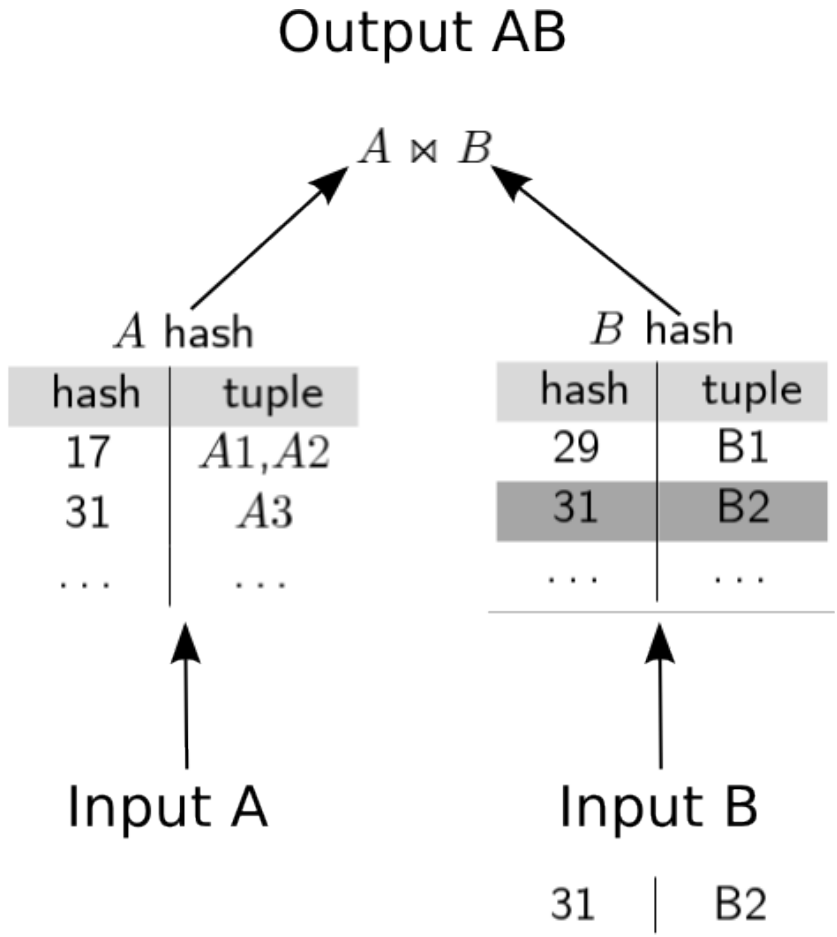
- “klassisches” Joinverfahren
- Grundlegende Idee $A \bowtie B$
  - Eine Eingaberelation (A) wird von einem Zwischenergebnis gelesen, die andere (B) in der Art einer Pipeline im Join verarbeitet
  - Alle Tupel von A werden in einer Hashtabelle gespeichert
    - Die Hashfunktion wird auf dem Joinattribut ausgewertet
      Alle Tupel, die den gleichen Wert für das Joinattribut haben, landen im gleichen Bucket
  - Jedes über die Pipeline eintreffende Tupel von B wird ebenfalls nach den Joinattribut gehasht
  - Das Tupel wird dann mit allen Tupeln von A im entsprechenden Bucket der Hashtabelle verglichen
  - Die Tupel mit übereinstimmenden Joinattributen landen im Ergebnis

Double-pipelined hash join

- Dynamischer Aufbau von Hashtabellen für Tupel aus *A* und *B*
- Verarbeite Tupel, wenn sie eintreffen
  - Cache Tupel, wenn es nötig ist
  - Verarbeite gleichmäßig Tupel aus *A* und *B*, um gute Performance zu erzielen
  - Bestimme ein gutes A:B Verhältnis auf Basis von Statistiken
- Wenn ein neues Tupel von Relation *A* eintrifft
  - Füge es in die Hashtabelle von *A* ein
  - Suche in der Hashtabelle von *B* nach Joinpartnern
  - Wenn es welche gibt, liefere alle kombinierten AB-Tupel als Ergebnis zurück
- Wenn ein neues Tupel von Relation *B* eintrifft, verarbeite es analog

Double-pipelined hash join – Beispiel

- $B(31, B2)$ kommt an
- Füge es in die Hashtabelle von B ein
- finde passende Tupel von A
  - $A3$ wird gefunden
  - Unter der Annahme, dass $A3$ zu $B2$ passt. . .
- Füge $AB(A3, B2)$ zum Ergebnis hinzu



Pipelining in verteilten Systemen

In Pipelines “fließen” die Tupel durch die Operationen

- Funktioniert sehr gut mit einer Verarbeitungseinheit (einem Rechner)
- Problem: Das einzelne Senden jedes Tupels von einem Rechner zum anderen ist in der Regel ineffizient
- Kommunikationskosten
  - Vorbereitung der Übertragung und Öffnen eines Kommunikationskanals
  - Zusammenstellen einer Nachricht
  - Übertragen der Nachricht: Kopfdaten und Nutzinformation (minimale Paketgröße ist größer als ein Tupel)
  - Empfangen und Zerlegen einer Nachricht
  - Schließen des Kanals

Pipelining in verteilten Systemen – Blöcke von Tupel

Minimiere den Kommunikationsoverhead durch das Packen von Tupeln in Blöcke

- Sende keine einzelnen Tupel, sondern Blöcke mit mehreren Tupeln
  - kurze, schnelle Kommunikation
  - Pakete müssen gecacht werden, bis sie vollständig sind
  - Blockgröße soll mindestens die Paketgröße des Netzprotokolls sein

Also noch mehr Kostenfaktoren im Kostenmodell

Zusammenfassung der globalen Anfrageoptimierung

Globale Anfrageoptimierung muss mit zusätzlichen Randbedingungen und Kostenfaktoren verglichen mit “klassischer” Anfrageoptimierung arbeiten

- Netzkosten, Netzmodell, Shipping-Regeln
- Methoden zur Fragmentierung und Allokation
- verschiedene Optimierungsziele (Antwortzeit vs. Gesamtbearbeitungszeit)