UNIVERSITÄT DES SAARLANDES

Saarbrücken, 09.07.2015
Information Systems Group

Vorlesung „Informationssysteme“

Vertiefung Kapitel 10: Sensor-Datenbanken

Erik Buchmann (buchmann@cs.uni-saarland.de)

Foto: M. Strauch

Aus den Videos wissen Sie...

- ...dass DMBS standardisiertes, deklaratives Datenmanagement bieten
  - aber auf klassische Client-Server-Architekturen zugeschnitten sind
- …dass DBMS auf relationalen Daten arbeiten
  - was ist mit ereignisbasierten Datenströmen?

- Vertiefung heute:
  - Einführung Drahtlose Sensornetze
  - Datenbank-Layer für Drahtlose Sensornetze

Einführung Sensornetze

Foto: M. Strauch

Wireless Sensor Networks (WSN)

- Netzwerke aus vielen, kleinen, billigen Rechnerknoten mit
  - unterschiedlichen Sensoren
  - drahtlosen Transmittern mit simplem Übertragungsprotokoll
  - wenig Speicher, wenig Rechenleistung
  - eng begrenzter Batteriekapazität
- Hochgradig verteilte Systemarchitektur
  - hunderte bis tausende Knoten
  - Kommunikation über Ad-Hoc-Routing
- *Aufgabe: einen (mehrere) komplexen Meßvorgang durchführen*

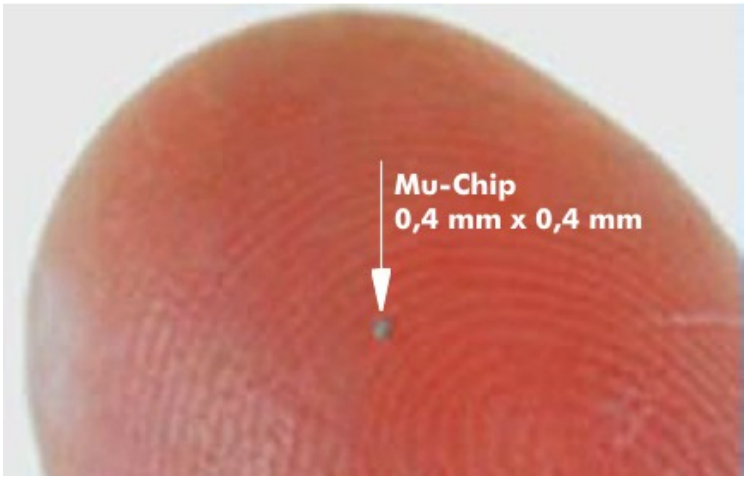


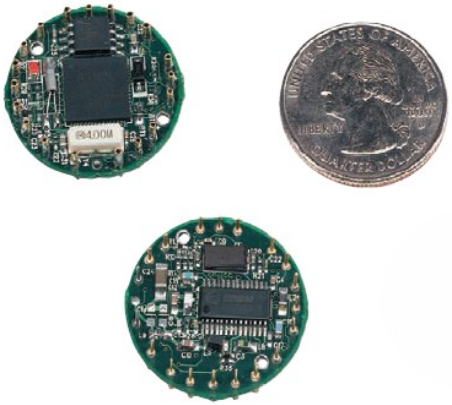
Bild: 2.45 GHz RFID-Chip, Hitachi

Beispiel 1: Mica Motes

- **Processor**: Atmel AT128L, 8Bit
- **Memory**: 4KB RAM, 128KB Flash
- **OS**: TinyOS
- **Programming**: NesC (C dialect)

Mica2 Sensor Nodes

- Mica2
  - 58x21x7 mm, 2 AA batteries
- Mica2DOT
  - 25mm diameter, 3V Coin Cell battery
- Unterschiedliche Sensor-Boards,
  z.B. für Temperatur, Licht, Feuchte,
  GPS, Magnetometer, Schall...



MICA2DOT Nodes

Beispiel 2: Sun SPOT (etwas veraltet...)

- **Processor**: ARM920T, 32-Bit
- **Memory**: 512KB RAM, 4MB Flash
- **OS**: Squawk VM (Java on bare metal)
- **Programming**: Java

- SunSPOT
  - SPOT = Small Programmable Object Technology
  - 62x37x25 mm, Lilo-Akku, 3.6V

- Eingebaute Sensoren: Licht, Temperatur, Beschleunigung

Anwendung: Habitat-Monitoring

Source: Zoological Society London

Anwendung: Lagerung von Gefahrgütern

Source: SAP, CoBis-Projekt

Anwendung: Energiewende

Forschungsziel: Richtig große WSNs

- Smart Dust: tausende Knoten müssen sich selbst organisieren
  - Katastropenmanagement
  - Umwelt-Monitoring in Nationalparks
  - Perimeterüberwachung, Einbruchserkennung
  - Klimaforschung, Ozeanographie, Vulkanologie
  - Bautechnik, Gebäudemanagement
  - ...

Beispiel: Katastrophenmanagement mit Smart Dust

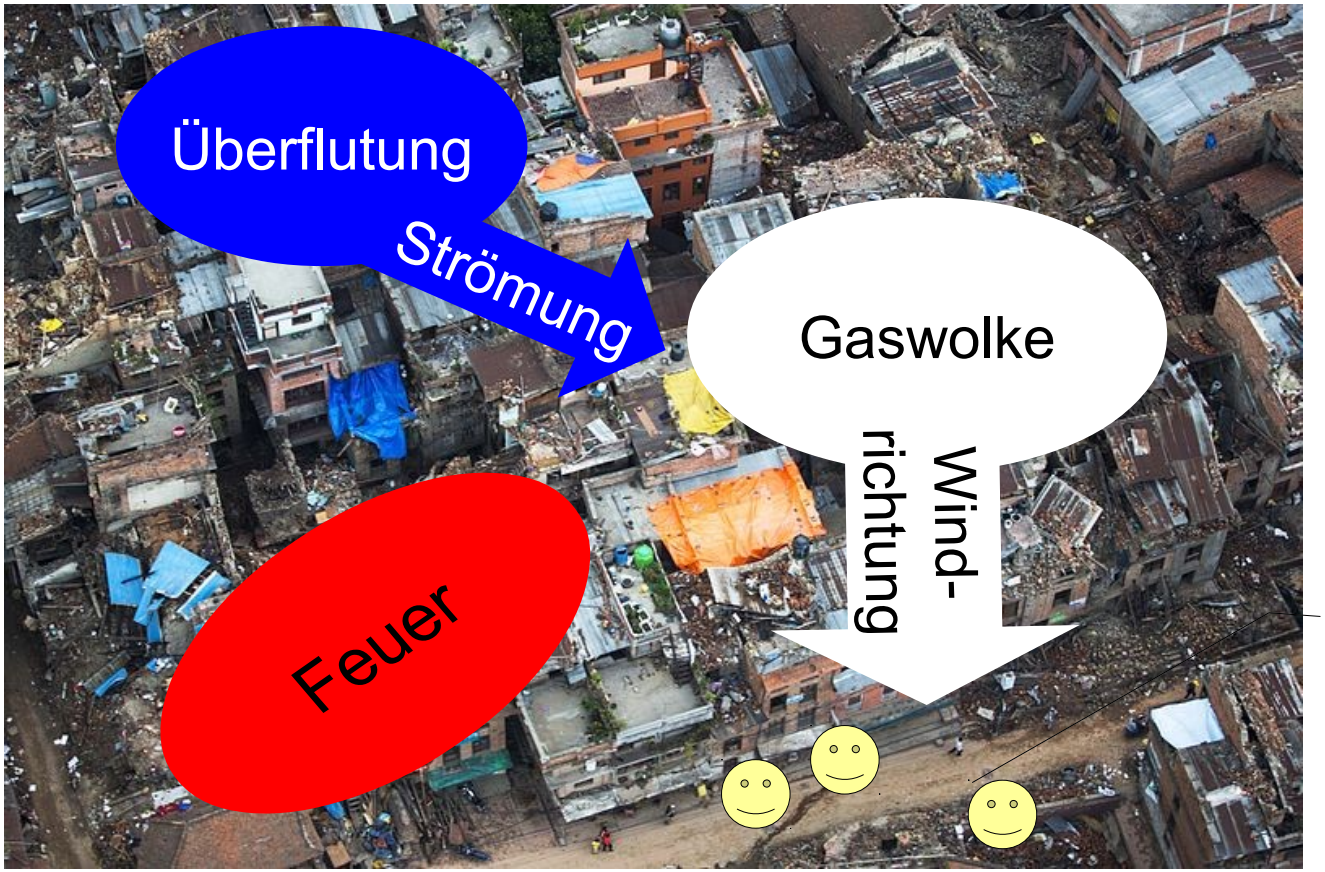
- Eine kleine, abgelegene Stadt in Nepal

Erdbeben!

- Nepal, April 2015

Ziel: Schnell wichtige Informationen erhalten

- komplexe Fragestellungen



1. Sensor Network Deployment

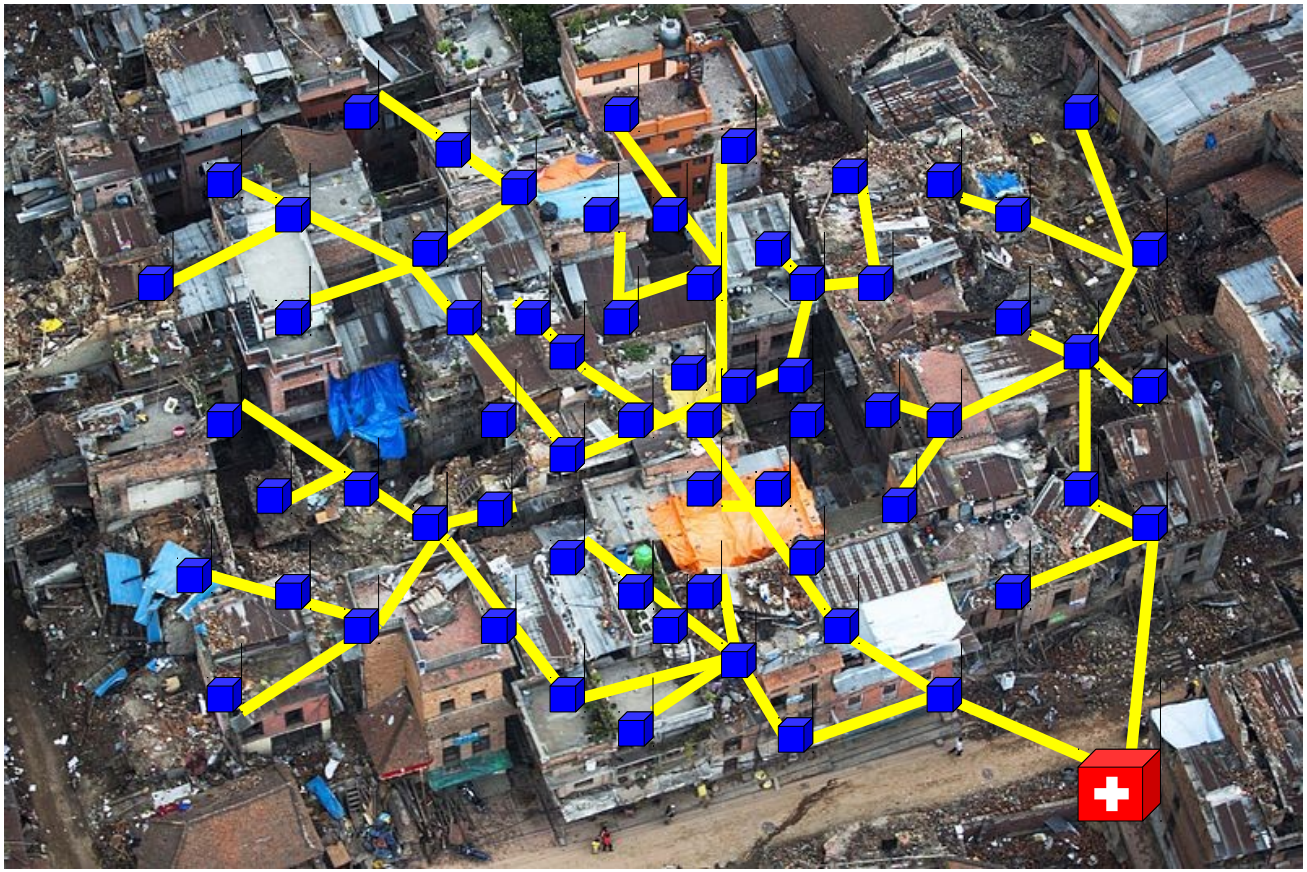
- Ausbringen der Knoten, z.B. Abwurf aus dem Flugzeug

1. Sensor Network Deployment

- Knoten an zufälligen Positionen, wissen nichts voneinander

2. Selbstorganisation der Knoten

- Aufbau eines Ad-Hoc-Netzwerks mit Routing-Baum zur Basis

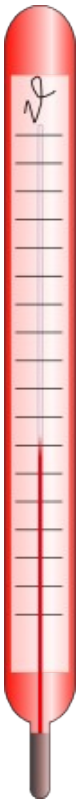


3. Query-Ausbringung

- Basis-Station sendet Anfragen an alle erreichbaren Knoten

4. Messung durchführen

- Knoten aktivieren Sensor-Ausstattung



5. Ergebnisse zur Basis zurückschicken

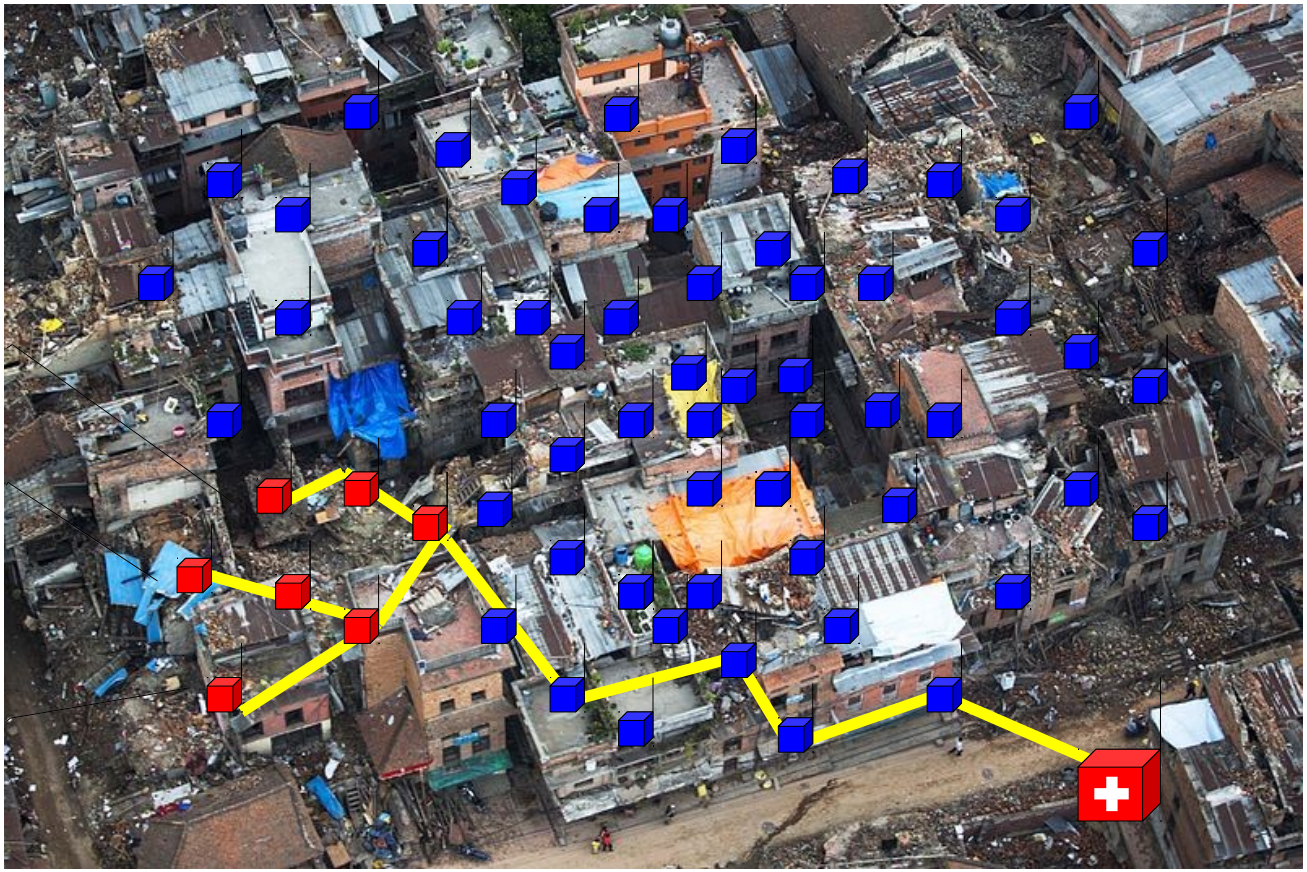
- Anfrage ausführen, Daten von Knoten zu Knoten weiterleiten

Pos.1, 250°C

Pos.2, 227°C

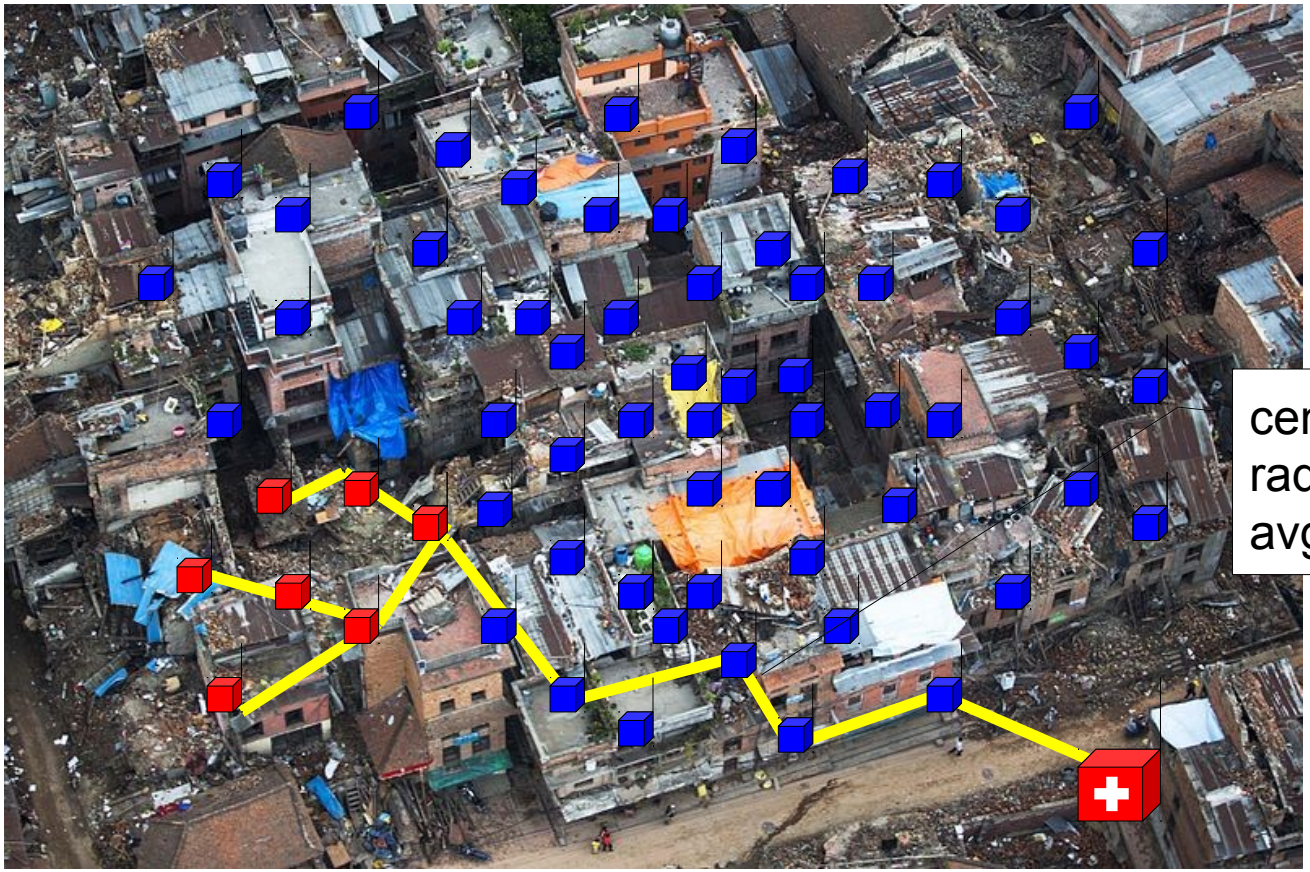
(...)

Pos.7, 375°C



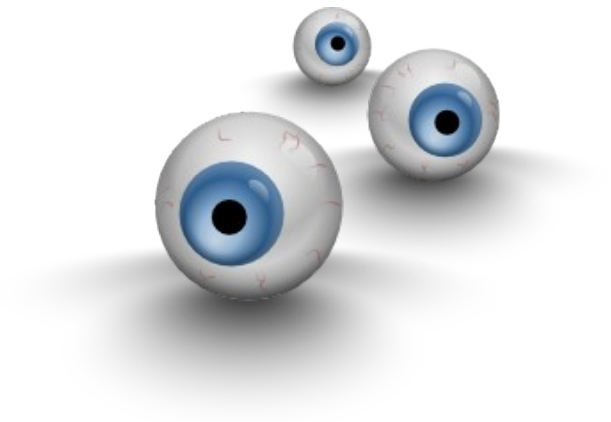
5. Ergebnisse zur Basis zurückschicken

- Daten im Netze aggregieren um Übertragungen zu sparen



Herausforderungen im WSN

- Hochgradig verteilte Systemarchitektur
  - jeder Knoten hat nur lokales Wissen
    → **Selbstorganisation**



- Begrenzte Ressourcen
  - Wenig Speicher, wenig Energie
  - Langsame CPU, Netzwerk
    → **Energie-effiziente Queries**
    (z.B. durch Verarbeitung an der Quelle)

- Unzuverlässigkeit
  - Häufige Ausfälle von Kommunikation oder einzelnen Knoten
    → **Best-effort Anfragen**
    (d.h., Ergebnisqualität nur so gut wie möglich)

WSN aus Datenbanksicht

- Warum sollte man eine Datenbanksicht auf ein WSN haben wollen?
  - Standardansatz für Datenmanagement
  - Deklaratives Query Processing
    - Anfragen unabhängig von der Technik und der Ausbringung
    - Anfrageoptimierung
  - Weiterverwendung existierender Werkzeuge für DBMS
    - Datenauswertung
    - Data Mining
    - Report-Generatoren
    - etc.

Anfrageverarbeitung in WSN

- Nicht einfach nur „das übliche“ + Techniken zum Energiesparen!
  - Knoten steuern, **wo, wann** und **wie viele** Daten erzeugt werden
    - keine a priori-Annahme dass Daten überhaupt existieren
  - Sensordaten sind ungenau
    - Sensoren und nachgeschaltete AD-Wandler mit begrenzter Auflösung
    - (zufällige) Ausbringung der Knoten entscheidet über Anfrageergebnis
  - *“**Acquisitional query processing**”*
- Gegensatz: herkömmliche DBMS gehen von exakten Daten aus
  - Korrektheitskriterien bezüglich der gespeicherten Werte

Was ist ein korrektes Anfrageergebnis?

- In herkömmlichen DBMS
  - Evaluierung der Anfrage auf DB-Zustand zu Transaktionsbeginn
  - Systemfehler haben keine Auswirkungen auf das Ergebnis
  - Transaktionale Garantien, insbes. Isolation Levels
    - z.B. „Read Committed“: korrektes Ergebnis trotz Nonrepeatable Reads

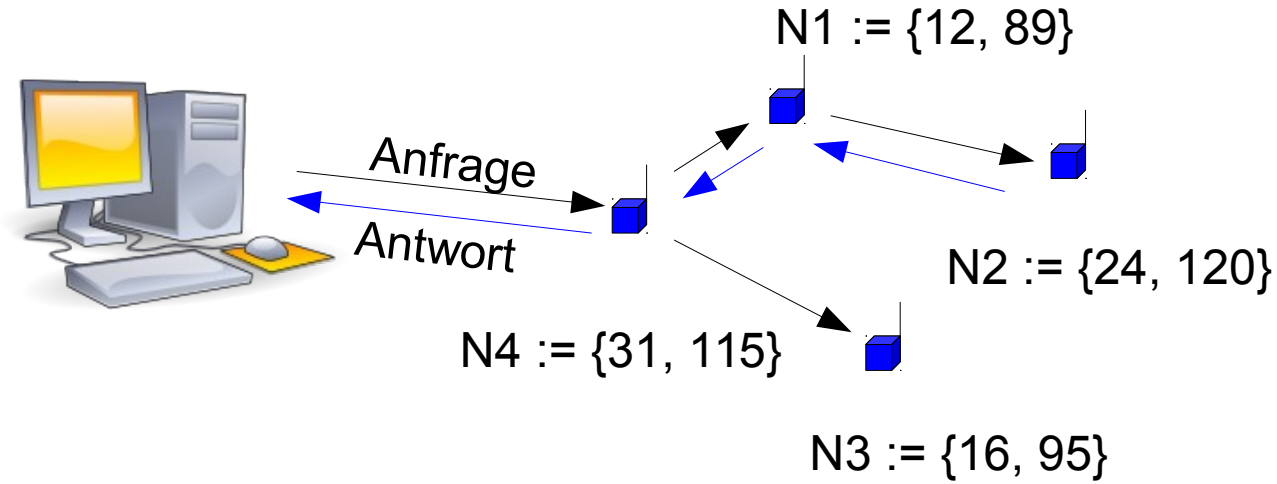
- In Wireless Sensor Networks (TinyDB, REED)
  - Evaluierung der Anfrage auf Daten, die alle Sensoren gleichzeitig messen
  - Knoten dürfen jederzeit ausfallen
  - Ausfall eines Knotens ohne Auswirkungen auf Anfrageauswertung
    - d.h., Knotenausfall ≡ kein Daten, NICHT: Knotenausfall ≡ Fehler (falsche Sortierreihenfolge, zuviele Tupel im JOIN-Ergebnis, etc.)

TinyDB: Aquisitional Query Processing

Foto: M. Strauch

Systemarchitektur

- Eine Basis-Station: Rechner ohne enge Ressourceneinschränkungen
  - Sicht auf das WSN: Virtuelle Relation „Sensors“
- Zahlreiche Knoten: begrenzte Ressourcen
  - Sicht auf das WSN: Nur lokal bzw alle Knoten in Funkreichweite



- Teuer: alle Daten zur Basis senden, dort Anfrageauswertung
- Im folgenden: Welche Operationen im Netzwerk ausführbar?

Virtuelle Sicht “Sensors”

- Jedes Tupel: Messwerte eines Sensors zu einer Zeit
- Attribute entsprechend der Sensorik, z.B. Temperatur, Licht, etc.
  - Attribute werden erst gemessen, wenn Query eintrifft
  - Attribute werden für Zeit der Anfrageverarbeitung gespeichert
- Besondere Attribute: NodeID, Timestamp
- Relation ist horizontal partitioniert über alle Knoten
- Zwischenergebnisse speicherbar in „Storage Points“

Sensors			
NodeID	**Time**	**Temp.**	**Light**
N1	T1	12	89
N2	T1	24	120
N3	T1	16	95
N4	T2	31	115

Unterstützte Anfragen

- Untermenge von SELECT-FROM-WHERE-GROUPBY-HAVING
  - Selektion, Projektion, Join (mit Einschränkungen)
  - Einige Aggregate (min, max, average)
  - Gruppierungen
- WSN-spezifische Erweiterungen
  - Kontinuierliche Anfragen für Stream-Verarbeitung
    - ONCE vs. SAMPLE PERIOD
  - lokale Zwischenspeicher für spätere Weiterverarbeitung
    - STORAGE POINT
  - Zeitliche Aggregate
    - WINAVG, WINMIN, ...
  - Ereignisbasierte Anfrageverarbeitung

Selektion, Projektion

- Jeder Knoten kann lokal auf seiner Partition von „sensors“
  - Attribute auswählen und
  - darauf mit Selektionsprädikaten filtern

```
SELECT NodeID, Temp
FROM sensors
WHERE Light > 100
ONCE;
```

  - Alle Knoten mit Light > 100 senden ein Tupel {NodeID, Temp} an die Basis
  - ONCE: Anfrage soll genau einmal ausgeführt werden, vgl. nächste Folie

Sensors

NodeID	Time	Temp.	Light
N1	T1	12	89
N2	T1	24	120
N3	T1	16	95
N4	T2	31	115

Kontinuierliche Anfragen auf Datenströmen

- Den Knoten mitteilen, wann und wie oft Daten erzeugt werden

```
SELECT nodeid, light
FROM sensors
SAMPLE PERIOD 1s FOR 10s;
```

  - Jeder Knoten sendet einmal pro Sekunde nodeid, light zur Basis-Station, und das 10 Sekunden lang

- Vorbedingung: globale Zeitsynchronisation
  - sonst erhält Basis-Station Daten aus unterschiedlichen Zeiten

Blockierende Operationen auf Datenströmen

- Def.: Eine Operation ist **blockierend**, wenn sie auf Tupel warten muss.
  - von anderen Knoten (symmetrische JOINs, nicht LEFT JOIN)
  - aus dem eigenen Stream (SORT, ORDER-BY, GROUP BY)

- Problem
  - Datenströme mit potentiell unendlicher Länge
  - Knoten mit begrenzter Speicher- und Übertragungskapazität
  - Knoten müssen jederzeit ausfallen dürfen
    → darf nicht die Ergebnisse von intakten anderen Knoten beeinflussen!

- Lösung in TinyDB
  - Ausschnitt eines Datenstroms („Window“) als STORAGE POINT speichern
  - Blockierende Operationen ausschließlich mit Storage Points

Storage Points

- Storage Point anlegen

```
CREATE STORAGE POINT st SIZE 8 AS (
        SELECT nodeid, light
        FROM sensors
        SAMPLE PERIOD 10s);
```

  - Jeder Knoten speichert lokal 8 Tupel {nodeid, light} lokal
  - alle 10 Sekunden neuen Wert einfügen und ältesten Wert löschen

- Blockierende Operationen nur erlaubt zwischen
  - zwei Storage Points auf dem gleichen Knoten
  - einem Storage Point und der lokalen Partition von „Sensors“

→ Funktionalität begrenzt
→ Knoten kann bei Fehler nicht die Werte von anderen „vergessen“

Beispiel

- JOIN zwischen Sensors und Storage Point „st“ von der letzten Folie

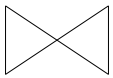
```
SELECT count(*) FROM sensors, st
WHERE sensors.nodeid = st.nodeid AND sensors.light < st.light
SAMPLE PERIOD 10s
```

- Rückgabe: ein Datenstrom von count(*)-Werten, die anzeigen wie oft die gemessene Helligkeit in den letzten 8 Samples höher war als die aktuell gemessene Helligkeit

auf jedem Knoten:

Sensors

NodeID	Time	Temp.	Light
N1	T10	12	89



st

NodeID	Light
N1	89
N1	120
N1	95
N1	76
N1	98
N1	132
N1	112
N1	115

Aggregation im WSN

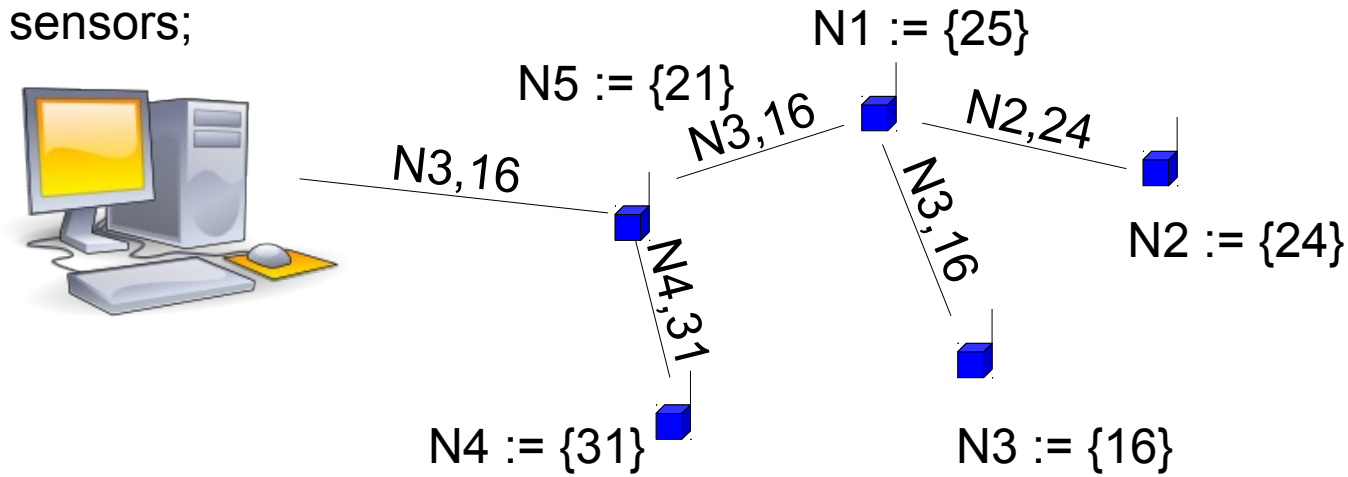
Foto: M. Strauch

Aggregation im WSN

- Funktionsprinzip: jeder Knoten
  - mißt selbst Daten und erhält weitere von seinen Kindern im Routing-Tree
  - berechnet daraus Zwischenergebnis
  - sendet Zwischenergebnis zu seinem Vater im Routing Tree

  - ***Welche Aggregatfunktionen können so berechnet werden?***

SELECT nodeid, MIN(temp)
FROM sensors;



Aggregationsfunktionen

- Klassifikation von Aggregatfunktionen
  - *distributiv*
  - *algebraisch*
  - *holistisch*

- Gegeben ein zweidimensionales Set von Werten $\{X_{ij} \mid i = 1,\ldots, n_j; j = 1,\ldots, m\}$
  - m Teilmengen von n_j-Werten
    - Beispiel: j := NodeID
      i := Light

      m = 3

      $n_1 = 3$
      $n_2 = 1$
      $n_3 = 2$

NodeID	Time	Light
N1	T1	89
N1	T2	120
N1	T3	95
N2	T3	76
N3	T2	98
N3	T3	132

Distributive Aggregatfunktionen

- Def.: Eine Aggregatfunktion F() ist **distributiv**,
  wenn es eine kumulative Aggregatfunktion G() gibt, so dass gilt:

  $F(\{X_{ij}\}) = G(\{F(\{X_{ij} \mid i = 1,\ldots, n_j\}) \mid j = 1,\ldots, m\})$

  - F() wird auf jede Teilmenge isoliert angewendet
    G() berechnet das Endergebnis

  - Beispiele
    - COUNT(*) — *F = count(), G = SUM()*
    - MIN, MAX, SUM — *G = F, z.B.: F = MIN(),G = MIN()*

  - Im WSN
    - jeder Knoten berechnet F auf seinen Daten und sendet Ergebniswert weiter an seinen Vaterknoten
    - jeder Knoten berechnet G auf den Daten seiner Kindknoten

    → *Distributive Aggregatfunktionen im WSN berechnbar!*

Algebraische Aggregatfunktionen

- Def.: Eine Aggregatfunktion F() ist **algebraisch**, wenn es zwei Funktionen G() und H() gibt, so dass gilt:

$$F(\{X_{ij}\}) = H(\{G(\{X_{ij} \mid i = 1,\ldots, n_j\}) \mid j = 1,\ldots, m\})$$

  - G() berechnet ein p-Tupel auf einer Teilmenge
    H() berechnet das Endergebnis

  - Beispiel
    - AVG() G := berechne <sum,count> einer Teilmenge
      H := berechne sum / count
    - Varianz, Standardabweichung, Top-N,...

  - Im WSN
    - Jeder Knoten berechnet G und sendet ein p-Tupel zum Vaterknoten
    - Basis-Station berechnet H auf den erhaltenen p-Tupeln

    → *Algebraische Aggregatfunktionen im WSN berechnbar!*

Holistische Aggregatfunktionen

- Def.: Eine Aggregatfunktion F() ist **holistisch**, wenn für deren Berechnung der gesamte Datenbestand erforderlich ist, d.h.,

  $F() = F(\{X_{ij} \mid i = 1,\ldots, n_j\})$

  - kein konstanter Speicherbedarf

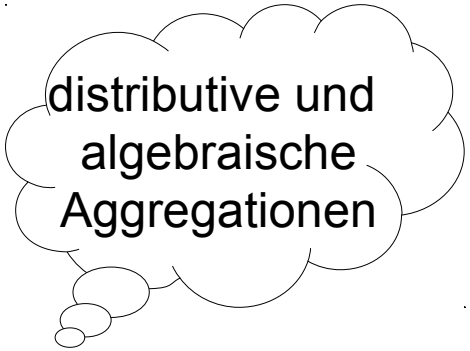
  - Beispiele
    - Median, Rank, MostFrequent, Quantil, Skyline

  - Im WSN
    - Jeder Knoten sendet seine Meßwerte zur Basis-Station

      → *Holistische Aggregatfunktionen im WSN teuer!*

Aggregation in TinyDB

- Jede Aggregation muss drei Funktionen implementieren
  - Merging Function f(), <z> = f(<x>, <y>)
    - <x>, <y>, <z> sind Tupel mit Teilergebnissen
  - Initializer i(), <x> = i(...)
    - Initialisiert ein Teilergebnis
  - Evaluator e(), r = e(<z>)
    - Berechnet das Endergebnis auf der Basis-Station



- Beispiel: AVG()
  - Teilergebnis ist <sum,count> einer Teilmenge
  - i(v) := <v,1>
  - f(<s_1,c_1>) = <s_1+s_2, c_1+c_2>
  - e(s,c) = s/c

Syntax von Aggregation / Gruppierung in TinyDB

- Identisch zu SQL
- Beispiel:

```
SELECT AVG(volume), room FROM sensors
        WHERE floor = 6
        GROUP BY room
        HAVING AVG(volume) > 10
        SAMPLE PERIOD 30s
```

  - Durchschnittslautstärke in den Räumen auf Etage 6, bei denen die Durchschnittslautstärke größer 10 ist.
    - Anm.: dazu muss jeder Knoten seine Werte für floor und room kennen

Temporale Aggregate

- Bis hierher: Aggregate über **gleichzeitig** gemessene Werte
  - Werte liegen auf verschiedenen Knoten
- Temporale Aggregate := Aggregate über die Zeit
  → Sliding Window Queries
  - in TinyDB: Entweder direkt über Storage Points, oder mit einer SQL-Erweiterung, die einen Storage Point anlegt
  - WINAVG, WINMAX, WINMIN, etc.
- Beispiel:

```
SELECT WINAVG(light, 30s, 5s)
FROM SENSORS
SAMPLE PERIOD 1s
```

  - Durchschnittshelligkeit jeweils über die letzten 30s, berechnet alle 5s auf Werten, die einmal pro Sekunde gemessen werden

Verteilter Verbund im WSN

Foto: M. Strauch

Innere und Äußere Verbundoperationen

LinkeRelation

A	B
1	2
2	3

RechteRelation

B	C
3	4
4	5

Natural Join

A	B	C
2	3	4

Outer Join

A	B	C
1	2	NULL
2	3	4
NULL	4	5

Left Outer Join

A	B	C
1	2	NULL
2	3	4

Right Outer Join

A	B	C
2	3	4
NULL	4	5

Verbund im WSN

- Wichtig für viele Anwendungen, z.B.
  - Klimaforschung: Was ist der minimalen Abstand zweier Punkte mit ähnlicher Temperatur?

```
SELECT MIN(distance(A.x, A.y, B.x, B.y))
FROM Sensors A, Sensors B
WHERE A.temp - B.temp > 15.0
```

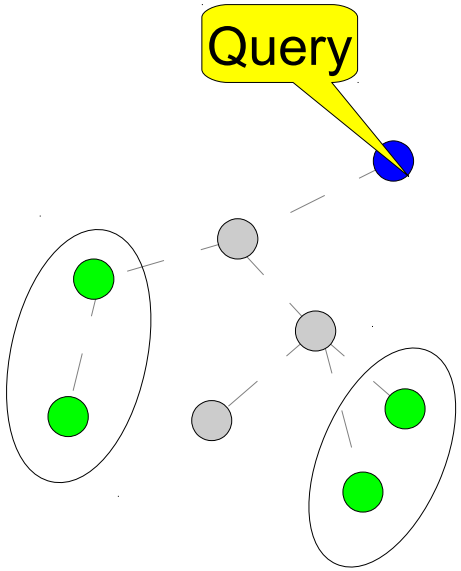
  - Umweltmonitoring: Was ist der Zusammenhang zwischen Luftfeuchte, Luftdruck und Temperatur?

```
SELECT |A.hum - B.hum|, |A.pres - B.pres|
FROM Sensors A, Sensors B
WHERE |A.temp - B.temp| < 0.3
AND distance(A.x, A.y, B.x, B.y) > 100
```

Verbundarten im WSN

- Drei Varianten sind relevant
  - **Selbstverbund bzw. Self Join**
    SELECT MIN(distance(A.x, A.y, B.x, B.y))
    FROM Sensors A, Sensors B
    WHERE A.temp - B.temp > 15.0
  - **Selbstverbund auf lokalen Daten**
    SELECT Count(*)
    FROM Sensors A, Sensors B
    WHERE A.id = B.id AND A.temp > B.light
  - **Verbund mit einer externen Relation**
    SELECT id, temp
    FROM Sensors A, ExtData B
    WHERE A.temp > B.temp AND A.hum > B.hum
- Anm.: alles andere kann auf der Basis berechnet werden

Was ist das Problem dabei?



- Knoten haben **nur lokale Sicht**
  - können **nicht entscheiden**,
    ob ihr temp-Wert auf anderen Knoten
    einen Verbundpartner findet

    SELECT *
    FROM Sensors A, Sensors B
    WHERE A.temp - B.temp > 15.0

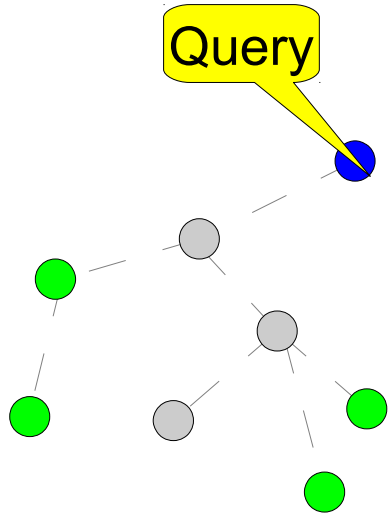
- Wo liegen die Tupel, die die Verbundbedingung erfüllen?
  A.temp - B.temp > 15.0

- Welche Daten wohin schicken, um Verbund energieeffizient zu berechnen?

Sensors

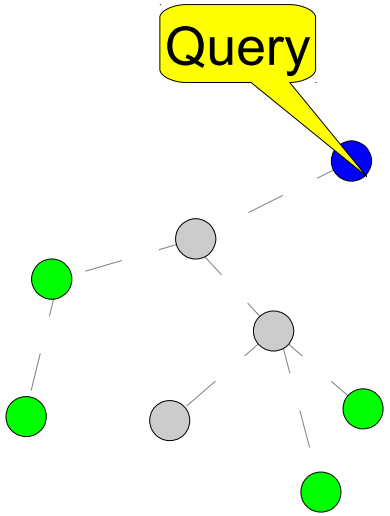
NodeID	Time	Temp.	Light
N1	T1	12	89
N2	T1	24	120
N3	T1	16	95
N4	T2	31	115

Wie JOIN im WSN berechnen?



- External Join → **funktioniert IMMER**
  - sende immer alle Daten zur Basis-Station
- TinyDB
  - nur Joins mit lokalen Daten
- REED
  - Join auf statischer Relation durch Broadcast
- SENS-Join
  - erstelle einen Filter, der bestimmt was zur Basis geschickt wird
- Local Join
  - sende eine Relation zur Position der anderen Relation
- Mediated Join
  - schicke alle Tupel zum Knoten auf dem Fermat-Punkt
- Local Semi-Join
  - Vorberechnung des Joins nur auf den Join-Attributen, Tupel später holen

Wo den Verbund berechnen?



- Extern bei der Basis-Station?
  - alle Daten zur Basis schicken kostet viel Energie
- Bei dem Knoten im WSN, zu dem sich alle Daten mit den geringsten Energiekosten übertragen lassen?
  - Wirtschaftsmathematikern als „Fermat-Punkt“ bekannt
  - Ist es billiger, alle Daten zum Fermat-Punkt zu senden und das Join-Ergebnis von dort zur Basis, als gleich alle Daten zur Basis zu schicken?
    - JOIN-Ergebnis kann größer sein als beide Relationen zusammen
- Lohnt es sich, Energie für das Sammeln von Statistiken aufzuwenden?
  - Mit ex-ante Wissen lässt sich eine optimale Verbund-Strategie bestimmen
  - ***Was ist eine gute Strategie ohne dieses Wissen?***

Entscheidung hängt ab von

- Verteilung der Knoten im WSN
  - Dichtbesetztes oder dünnbesetztes Netzwerk, Pfadlänge zur Basis
- Verteilung der Tupel über die Knoten
  - Zu Knoten sind die Übertragungskosten aller Daten minimal?
- Verbund-Operation
  - Insbes. symmetrisch oder asymmetrisch
- Verbund-Prädikate
  - Z.B. Equi-Join, Theta-Join
- Zahl der Verbund-Attribute
  - Wie umfangreich sind die zu übertragenden Tupel?
- Selektivität des Verbunds, Kardinalität der Relationen
  - Datenvolumen des Verbund-Ergebnisses größer als die Einzelrelationen?

Intra-Node Join (TinyDB)

vgl. Folie 33

- Join zwischen sensor-Relation und Storage Point st

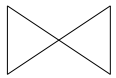
```
CREATE STORAGE POINT st SIZE 8 AS (
        SELECT nodeid, light FROM sensors
        SAMPLE PERIOD 10s)

SELECT count(*) FROM sensors, st
        WHERE sensors.nodeid = st.nodeid
        AND sensors.light < st.light
        SAMPLE PERIOD 10s
```

auf jedem Knoten:

Sensors

NodeID	Time	Temp.	Light
N1	T10	12	89



st

NodeID	Light
N1	89
N1	120
N1	95
N1	76
N1	98
N1	132
N1	112
N1	115

Join mit statischer Relation (REED)

- Beispiel

```
SELECT s.nodeid, a.condition
FROM sensors s, alert_table a
WHERE s.temp > a.temp
      AND s.humidity > a.humidity
      AND s.time = a.time
```

alert_table

condition	time	temp	humidity
1	9pm	20°C	70,00%
2	10pm	25°C	80,00%
3	11pm	30°C	90,00%
...			

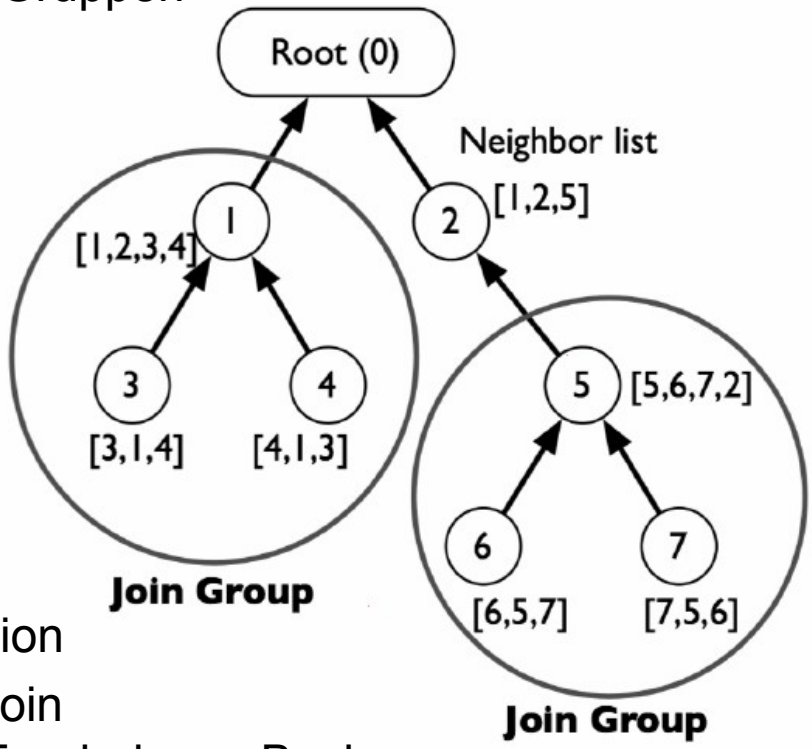
- Idee: sende eine statische Relation an alle Knoten im Netzwerk
  - Was passiert, wenn die statische Relation zu groß ist, um auf einem Knoten gespeichert zu werden?

Join with Large Static Relations (REED)

1) Group formation
   - Knoten in Broadcast-Reichweite bilden Gruppen

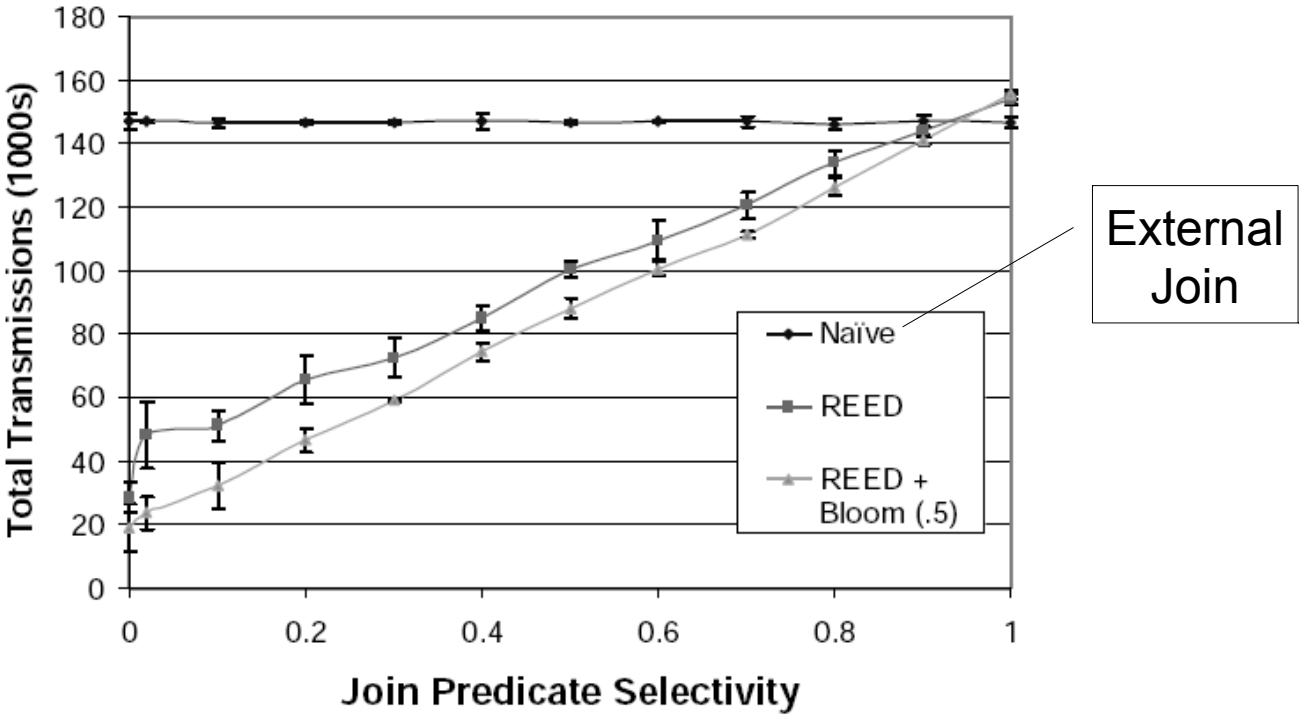
2) Table distribution
   - bestimme horizontale Partitionierung der statischen Relation
   - jeder Knoten einer Gruppe speichert eine Partition

3) Query processing
   - jeder Knoten einer Gruppe macht einen Broadcast mit seiner Partition
   - jedes Gruppenmitglied berechnet den Join mit dem empfangenen Daten, senden Ergebnis zur Basis
   - Knoten ohne Gruppe senden Rohdaten zur Basis



Performance von REED

- effizient, wenn Selektivität des Joins hoch und statische Relation klein



SENS-Join

- Ziel: Ausführung, die ohne ex-ante Wissen fast immer effizient ist

```
SELECT A.light - B.light
FROM Sensors A, Sensors B
WHERE A.temp - B.temp > 15.0
```

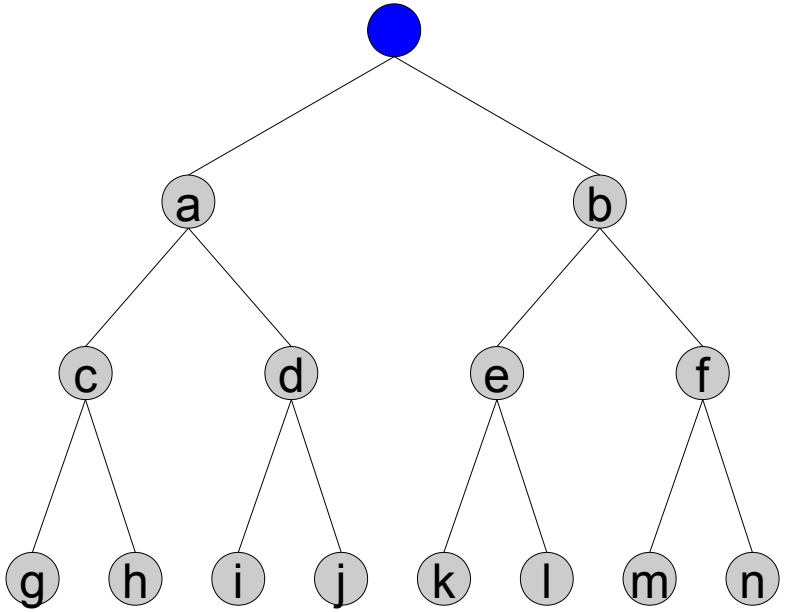
- Vorgehensweise
  1) Knoten senden alle Join-Attribute zur Basis-Station
     {12, 24, 16, 31}
  2) Basis berechnet Verbund
     $\{12, 24, 16, 31\} \rhd\lhd_{a\text{-}b>15} \{12, 24, 16, 31\}$
  3) Basis sendet Filter an alle Knoten
     {12|31}
  4) Knoten senden nur die Tupel zur Basis, die im Endergebnis enthalten sind
     N1: {12, 89}, N4: {31, 115}

Sensors

NodeID	Time	Temp.	Light
N1	T1	12	89
N2	T1	24	120
N3	T1	16	95
N4	T1	31	115

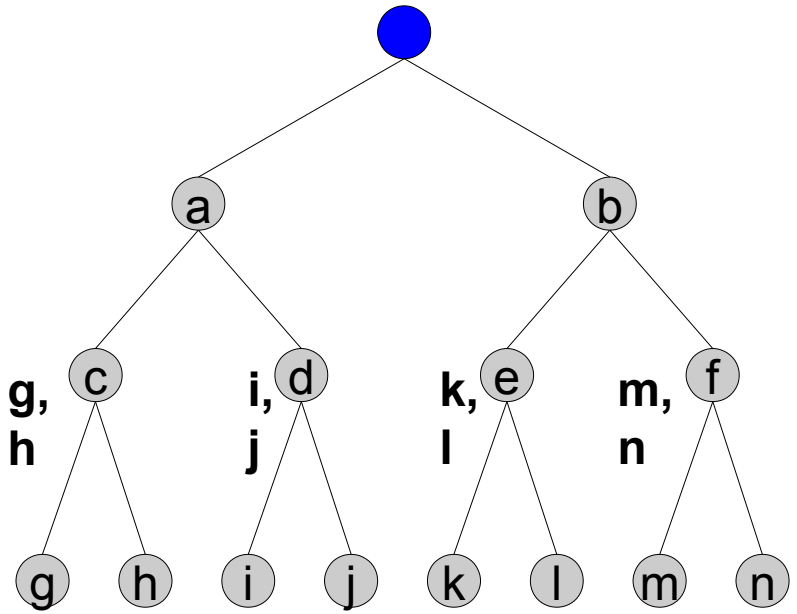
1. Join-Attribute zur Basis schicken

- Ausgangslage: Routing tree bereits erstellt, Anfrage ins WSN gesendet
- Jeder Knoten
  - sendet seine Join-Attribute Richtung Basis
  - sendet dabei vollständige Tupel, falls Datenpakete nicht voll
  - speichert Tupel seiner Kinder zwischen, falls Platz im Speicher



1. Join-Attribute zur Basis schicken

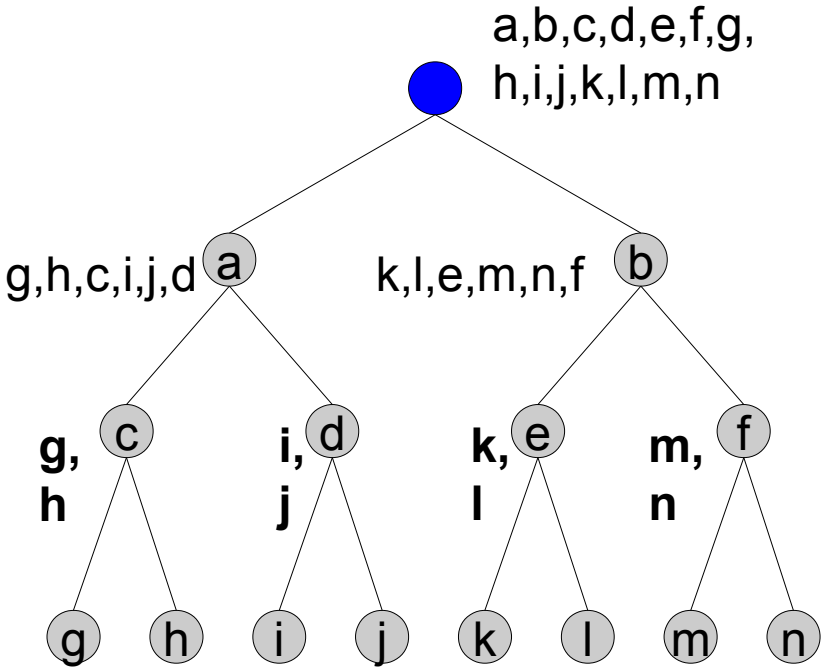
- Jeder Knoten
  - sendet seine Join-Attribute Richtung Basis
  - sendet dabei vollständige Tupel, falls Datenpakete nicht voll
  - speichert Tupel seiner Kinder zwischen, falls Platz im Speicher



*Anm: **Fett** dargestellte Buchstaben bezeichnen vollständige Tupel*

1. Join-Attribute zur Basis schicken

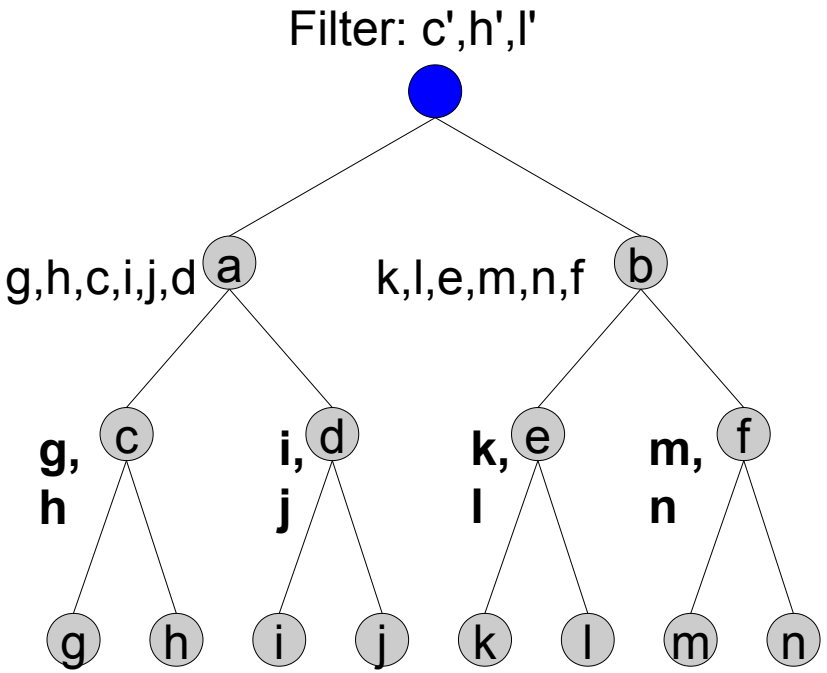
- Jeder Knoten
  - sendet seine Join-Attribute Richtung Basis
  - sendet dabei vollständige Tupel, falls Datenpakete nicht voll
  - speichert Tupel seiner Kinder zwischen, falls Platz im Speicher



*Anm: **Fett** dargestellte Buchstaben bezeichnen vollständige Tupel*

2. Filter berechnen

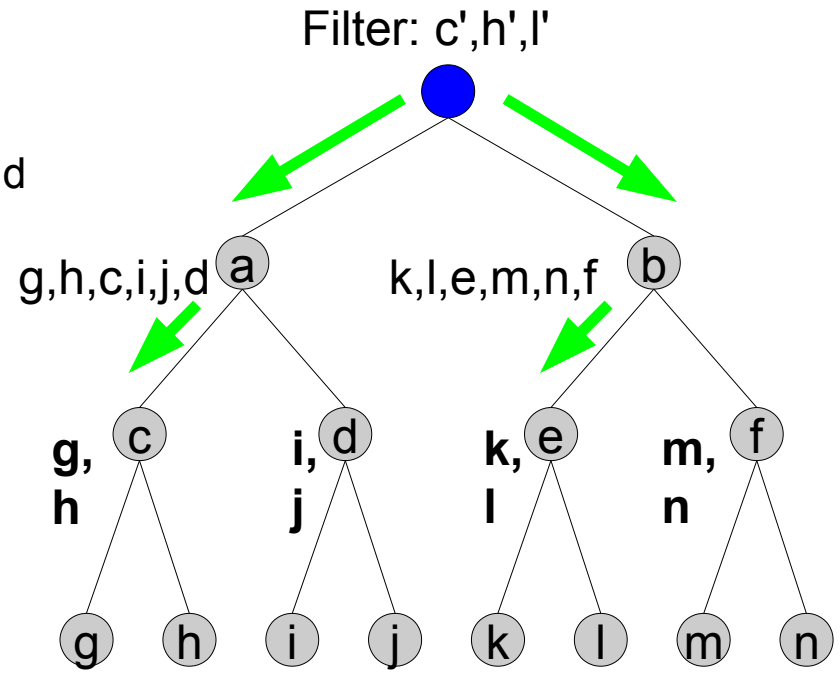
- Basis-Station
  - berechnet Verbund aus den erhaltenen Verbund-Attributen
  - berechnet einen kompakten Filter



*Anm: **Fett** dargestellte Buchstaben bezeichnen vollständige Tupel*

3. Filter an die Knoten schicken

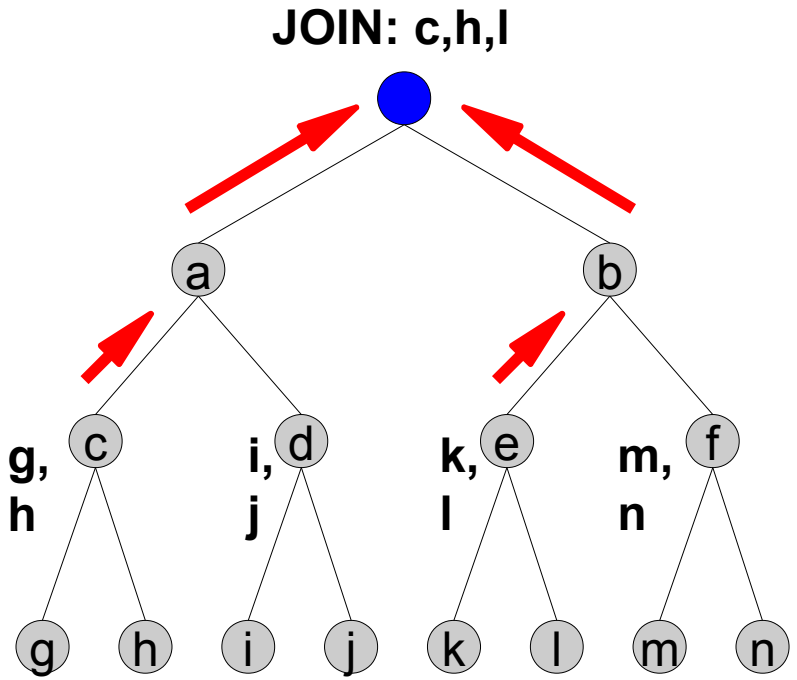
- Jeder Knoten
  - sendet den Filter an diejenigen seiner Kinder weiter, die
    - nicht im Zwischenspeicher sind
    - passende Tupel haben



*Anm: **Fett** dargestellte Buchstaben bezeichnen vollständige Tupel*

4. Tuples zur Basis schicken

- Jeder Knoten im Filter
  - sendet alle Query-Attribute zur Basis-Station
- Basis-Station
  - berechnet Endergebnis

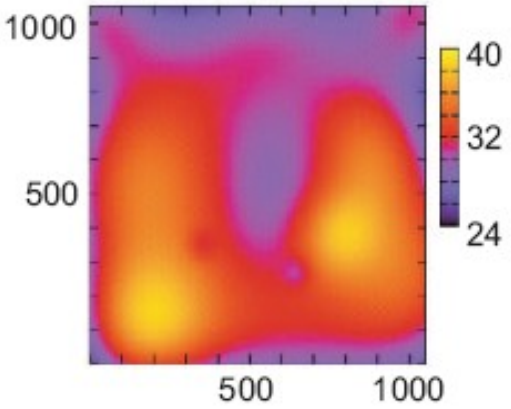


*Anm: **Fett** dargestellte Buchstaben bezeichnen vollständige Tupel*

Performanz von SENS-Join

- NS-2 Network Simulator
- Zufallsverteilung von 1500 Knoten auf 1km² Fläche
- 50m Kommunikationsreichweite, bidirektionale Übertragung

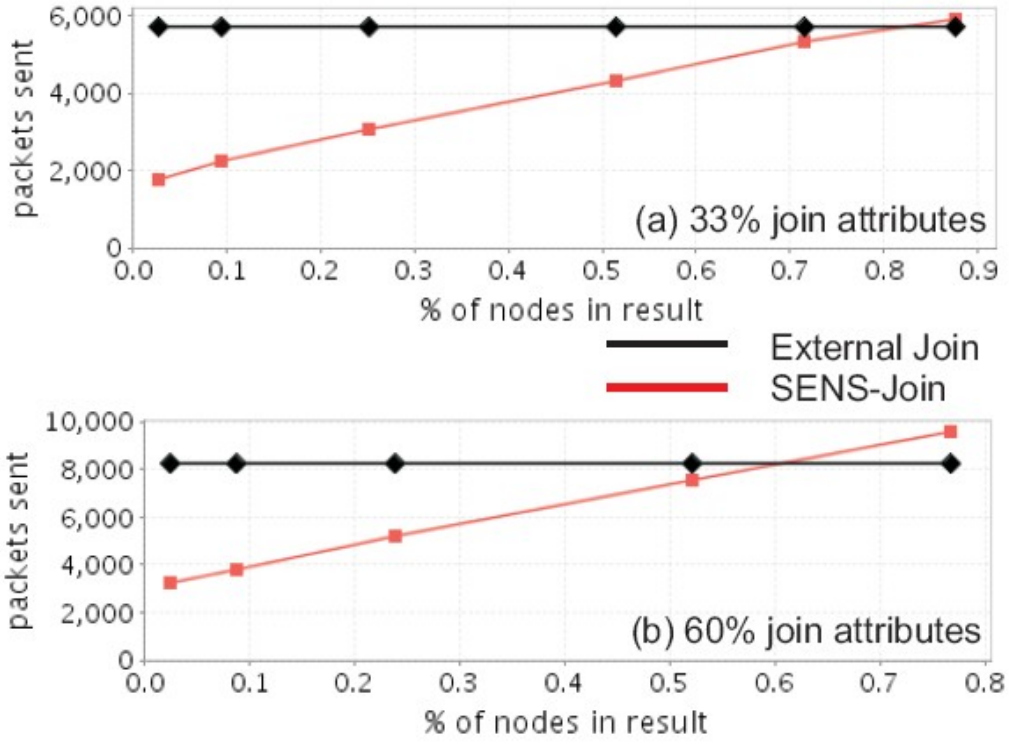
- Simulierte Meßwerte entsprechen der Grafik:



temperature

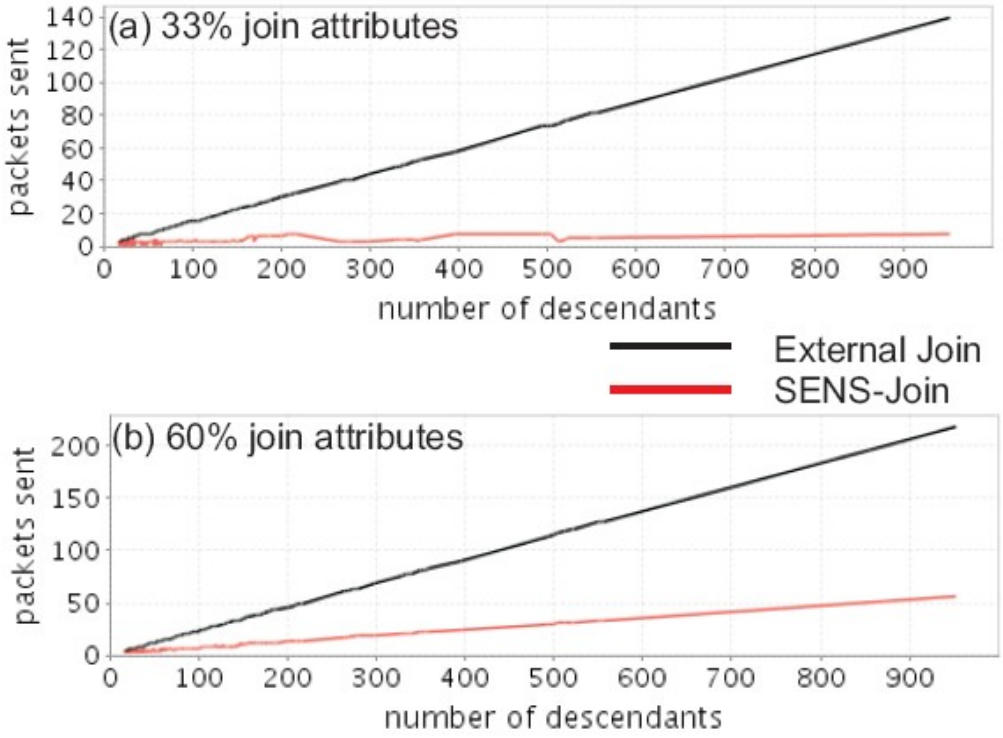
Ergebnisse (1/2)

- SENS-Join arbeitet besonders energieeffizient, wenn
  - die Selektivität des Verbunds hoch ist
  - viele Attribute in den Verbundprädikaten gebraucht werden



Ergebnisse (2/2)

- SENS-Join kommt auch mit tiefen Routing-Trees sehr gut zurecht (dünnbesetzte Netze mit langen Pfaden zur Basis)



(Selektivität 5%, d.h., 5% aller Tupel werden zur Basis geschickt)

Zum Abschluss

Foto: M. Strauch

Literatur

- S. MADDEN, M. FRANKLIN, J. HELLERSTEIN, W. HONG. *TinyDB: An Acquisitional Query Processing System for Sensor Networks*. In: ACM Transactions on Database Systems, 2005
- A. COMAN, M. NASCIMENTO, J. SANDER. *On Join Location in Sensor Networks.* In: Proceedings of the MDM 2007
- MIRCO STERN, ERIK BUCHMANN, AND KLEMENS BÖHM. *Towards Efficient Processing of General-Purpose Joins in Sensor Networks.* In: Proceedings of the ICDE, March 2009.

Wie geht es weiter?

- bis Montag, 13.07., 12 Uhr
  - Quiz: Mapreduce
- Dienstag, 14.07., GHH 12-14 Uhr: Tutoriumstermin
  - kurze Besprechung von Aufgabenblatt 10
  - nächstes Aufgabenblatt: Normalformen
- Donnerstag, 16.07.: Präsenztermin
  - Anfrageverarbeitung auf verschlüsselten Daten