Speicher- und Datenbanktechniken

Seminarvortrag von Patrick Leuthold

Betreuer: Michael Rohs

Übersicht

1. Motivation
2. Queryabarbeitung
3. Data Centric Storage
4. Tiny Aggregation

Sensornetz als Datenbank

Eine Datenbank (DBMS) befreit den Nutzer von folgenden Aufgaben:

-Speicherung (Persistenz)

-Operationen (erzeugen, ändern, speichern, suchen und entfernen von Daten)

-Sichten

-Integritätssicherung

-Zugriffskontrolle

-Transaktionen

Kritische Faktoren einer Datenbank

-Latenz

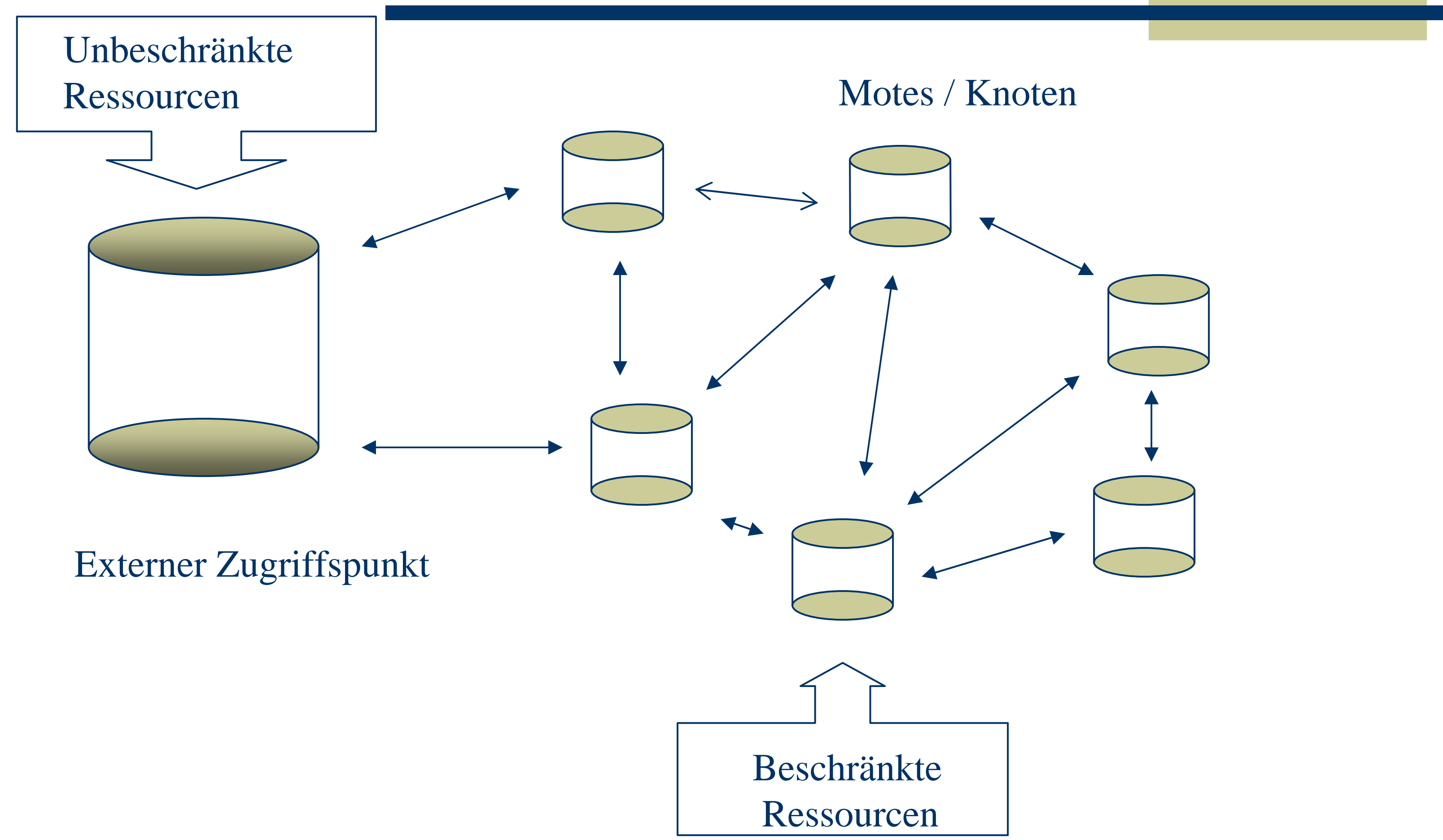
-Durchsatz

Zusätzlich für Sensornetze:

-Energieverbrauch

-Qualität der Daten

Verteilte Datenbanken



Motes

Prozessor	4 Mhz, 8 bit MCU (ATMEL)
Speicher	512KB
Funk	916Mhz Radio
Reichweite	30m
Bandbreite	40 Kbits/sec
Stromverbrauch Senden	12 mA
Stromverbrauch Empfangen	1.8 mA
Stromverbrauch CPU	5 mA
Stromverbrauch EEPROM	3mA

(Query Processing for Sensor Networks Yong Yao Cornell University)

Querys in Sensornetzen

Wenn Sensornetze Datenbanken sind, dann möchte man auch gerne auf diese wie in einer herkömmlichen Datenbank mit SQL darauf zugreifen können.

Historische Abfragen

Für das Jahr 2002 alle Punkte der Schweiz, die eine Durchschnittstemperatur von mehr als 24°C hatten.

Snapshot

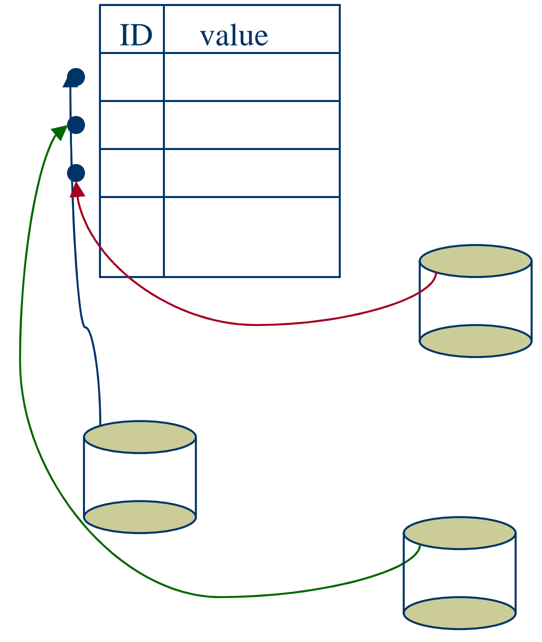
Welche maximale Temperatur haben wir jetzt in Zürich.

Querys in Sensornetzen

Sensornetze machen aus der Welt eine einzige Datenbank

Bestehende SQL Sprache erweitern

SELECT	{attributes, aggregates}
FROM	{Sensordata S}
WHERE	{predicate}
GROUP BY	{attributes}
HAVING	{predicate}
DURATION	time interval i
EVERY	time span e



S ist eine Virtuelle Tabelle.

Querys in Sensornetzen

Historische Abfragen

Für das Jahr 2002 alle Punkte der Schweiz die eine Durchschnittstemperatur von mehr als 24°C hatten.

```
SELECT AVG(s.temp), s.id
FROM Sensor s
WHERE s.generated > 1-1-2002 AND s.generated < 31-12-2002
GROUP BY s.id
```

Querys in Sensornetzen

Snapshot
Welche maximale Temperatur haben wir jetzt in Zürich.

```
SELECT MAX(s.temp)
FROM Sensor s
WHERE s.location = „zurich“
```

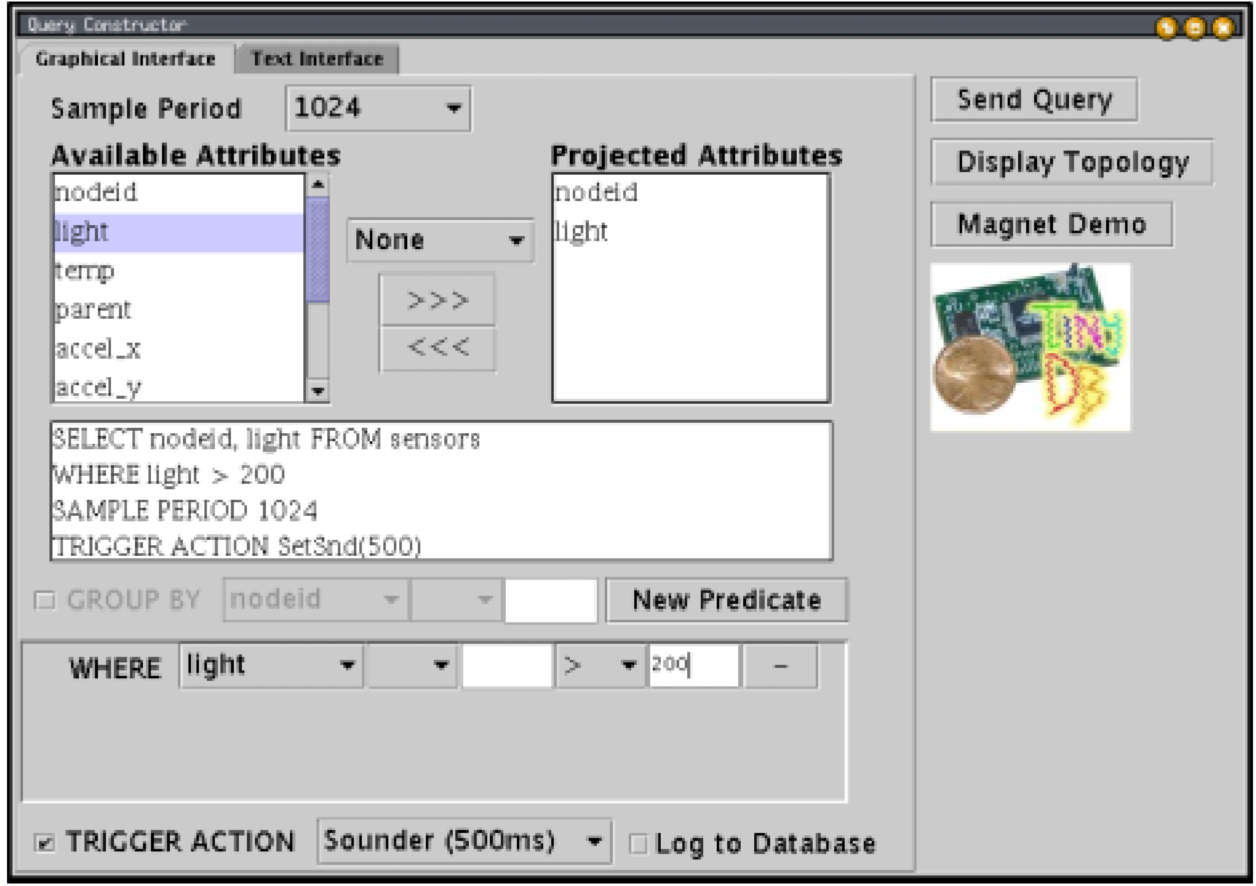
Querys in Sensornetzen

Laufende

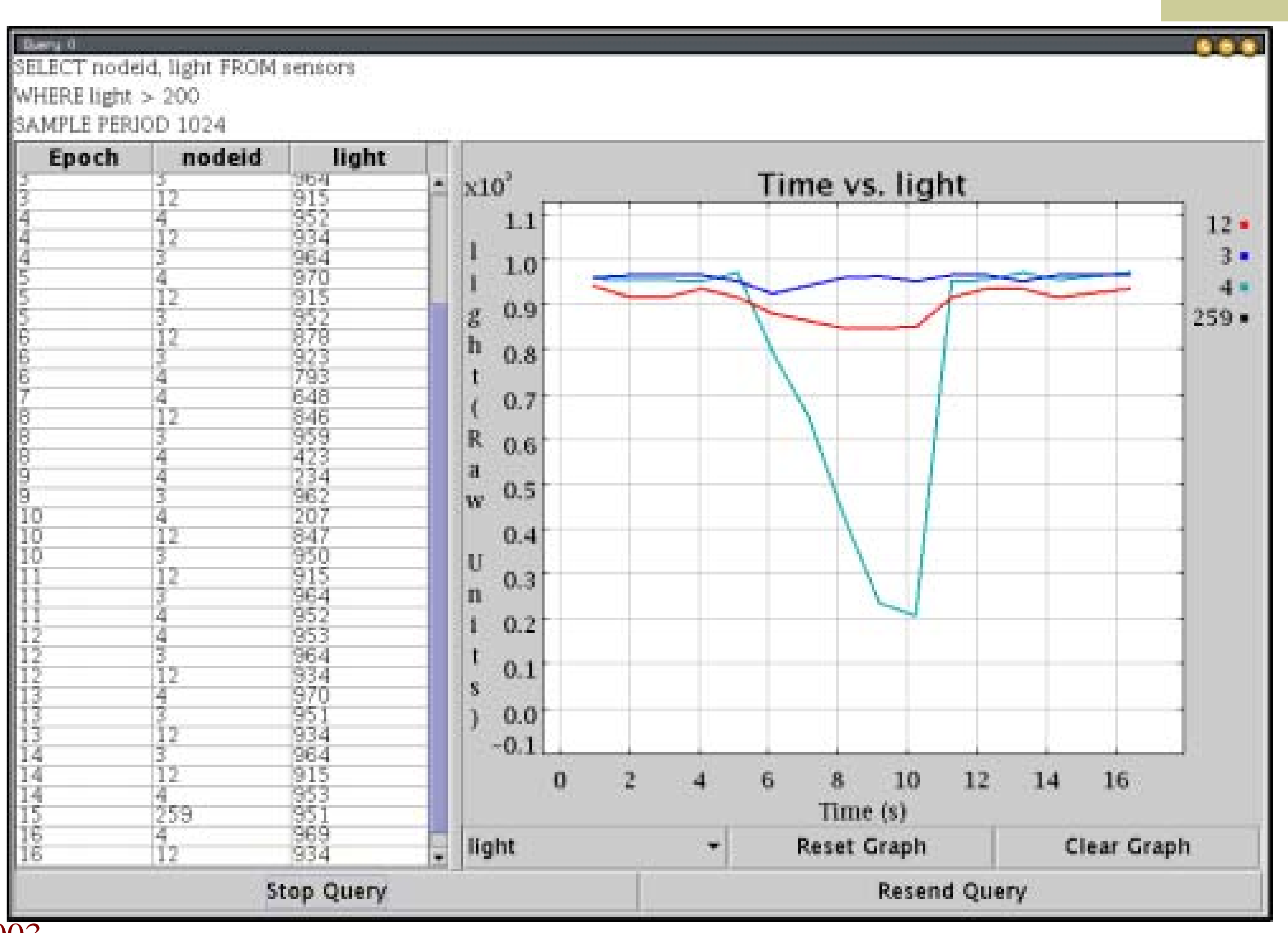
Für die nächste halbe Stunde möchte ich alle 30s die durchschnittliche Temperatur aller Sensoren in Zürich.

```
SELECT AVG(s.temp)
FROM Sensor s
WHERE s.location = „zurich“
DURATION (now, now + 1800)
EVERY  30
```

TINY DB



TINY DB



Epoch	nodeid	light
3	3	964
3	12	915
4	4	952
4	12	934
4	3	964
5	4	970
5	12	915
5	3	952
6	12	878
6	3	923
6	4	793
7	4	648
8	12	846
8	3	959
8	4	423
9	4	234
9	3	962
10	4	207
10	12	847
10	3	950
11	12	915
11	3	964
11	4	952
12	4	953
12	3	964
12	12	934
13	4	970
13	3	951
13	12	934
14	3	964
14	12	915
14	4	953
15	259	951
16	4	969
16	12	934

Übersicht

1. Motivation
2. Queryabarbeitung
3. Data Centric Storage
4. Tiny Aggregation

Verteilte Datenbanken

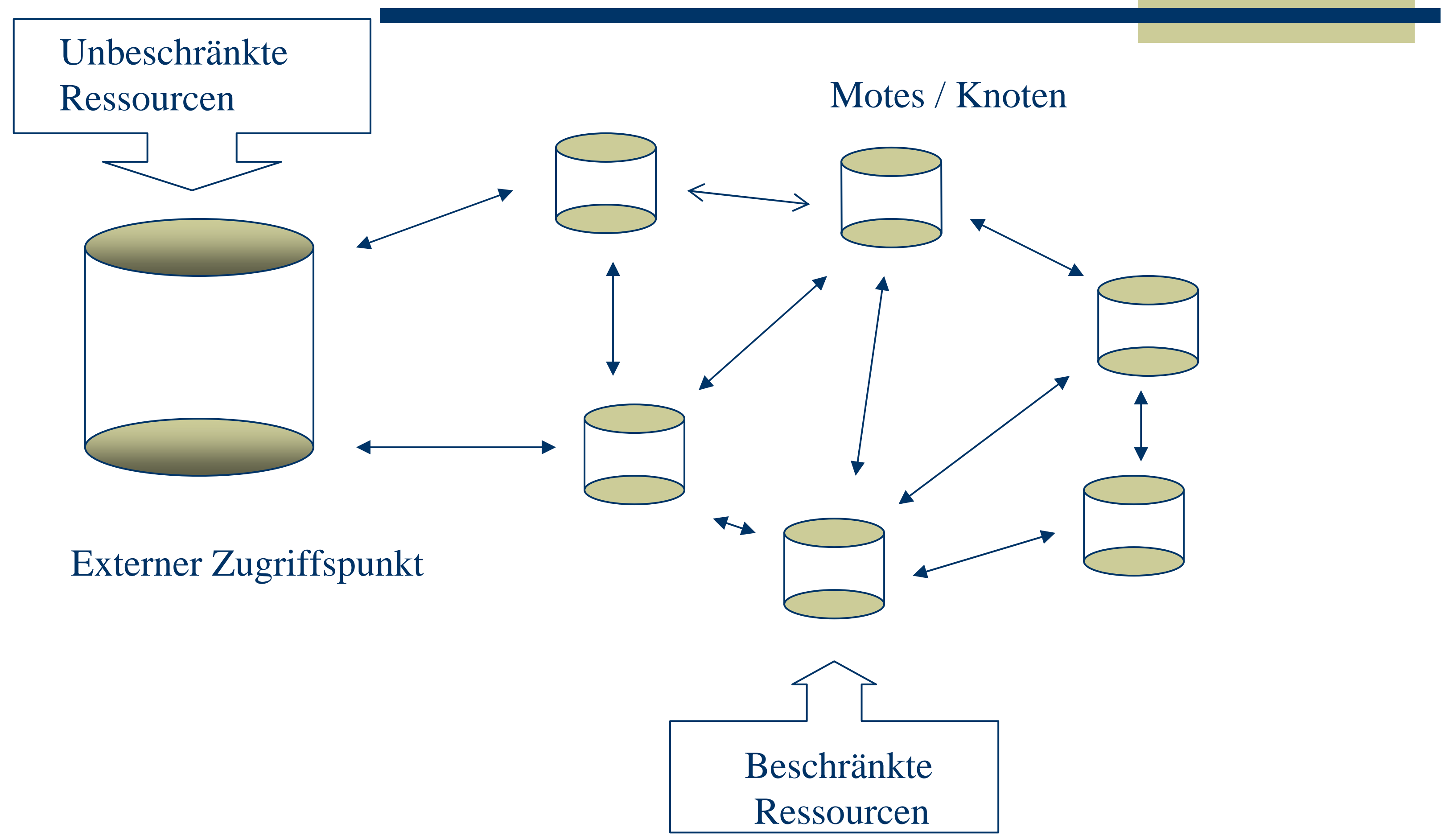
Folgende Query wird gestellt:

Alle Knoten, die eine Temperatur von mehr als 25°C haben mit ihren Ortsangaben

```
SELECT VR.value, R.x, R.y
FROM RFSensor R, VRFSensors VR
WHERE VR.temperatur > 25 AND VR.SensorID = R.SensorID
DURATION (now, now + 3600)
EVERY 30
```

VRFSensors ist eine Virtuelle Tabelle, in welche die Messresultate eingefügt werden.
R ist eine extern gespeicherte Tabelle mit Zusatzinformation über die einzelnen Sensoren.
Zwei Operationen: Selektion, Join Operation, (Projektion)

Aufbau eines Sensornetzes



Arbeitsteilung

Verschieden Möglichkeiten diese Query abzuarbeiten:

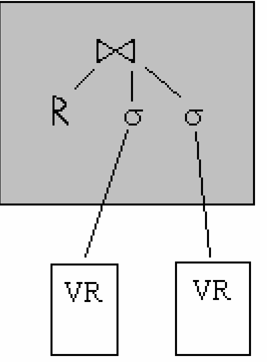
Möglichst viel extern rechnen:
+ kein Stromverbrauch für Rechenarbeit auf Knoten
- viele Nachrichten

⇔

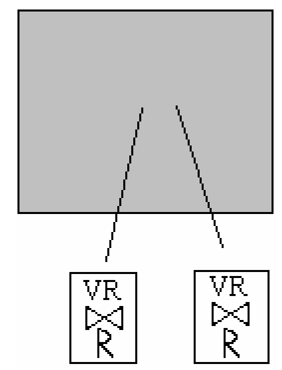
Möglichst viel auf den Knoten rechnen
+ weniger Nachrichten
- Stromverbrauch für Rechenleistung
- Initialisierungskosten

Query Abarbeitung

Alle Arbeit extern erledigen
Jeder Knoten schickt seine Messwerte an den externen Rechner.
Auf den externen Rechner wird die Selektion und die Join Operation durchgeführt.

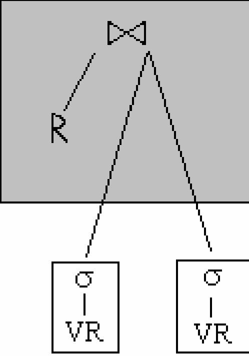


Alle Arbeit auf den Knoten erledigen
Der externe Rechner sendet jedem Knoten die Tabelle R.
Knoten führen Join und Selektion durch. Und senden die Resultate dem externen Rechner.



Query Abarbeitung

Selektion auf Knoten, Join auf externen Rechner
Die Selektion wird auf den Knoten durchgeführt, und der Join auf dem externen Rechner. Die Knoten senden nur die Messwerte, die der Selektion entsprechen.



Kritik

Alles Extern:

- Es werden viele Nachrichten geschickt.

Alles auf den Knoten

+ Es müssen nur die Messungen geschickt werden, die gebraucht werden.
- Hohe Initialisierungskosten
- Hohe Kosten für Join Operation

Geteilt

+ Weniger Nachrichten als bei erster Variante
+ Keine Initialisierungskosten
- Einige Messwerte werden umsonst geschickt

Test

Es wurde eine Simulation mit folgenden Annahmen durchgeführt:

Wcpu	0.000001 J/Instruktion
Wram	0
Wsend	0.059 J/msg
Wbdw	0.23 J/Kbytes
Kosten pro Operation (Join)	5’000

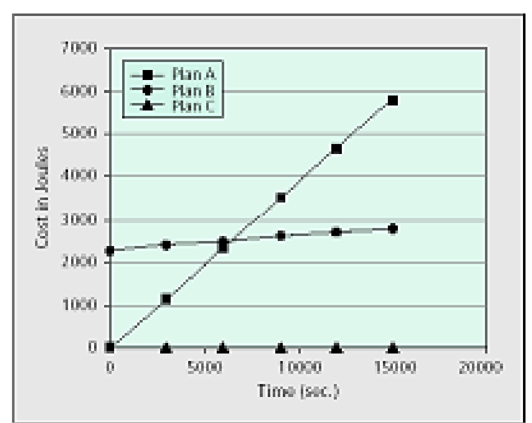
Kosten pro Operation: 5000 Instruktionen
Grösse pro Tupel: 50 Byte
200 Knoten insgesamt
Jeder Knoten direkt (!) mit dem externen Rechner verbunden

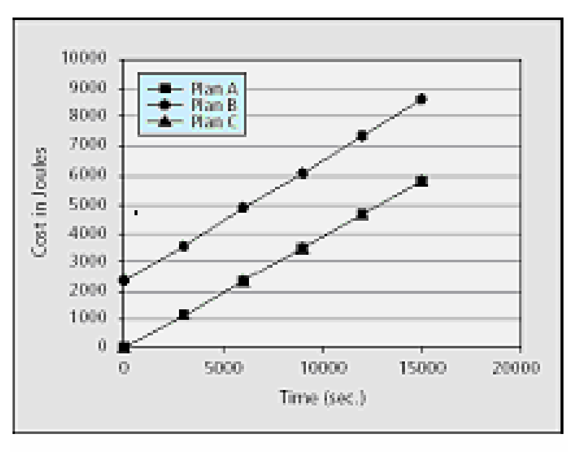
Test

Zwei Simulationen:

Kein Messwert über 25°C

Jeder Messwert über 25°C





Jedes Tupel in der Virtuellen Tabelle hat ein Gegenstück in der externen Tabelle => Join trägt nicht zur Reduktion der Nachrichten bei

Query Abarbeitung

Die richtige Wahl der Query Abarbeitung hängt von der Umgebung ab

Extern macht nie Sinn.

Die Wahl zwischen *alles auf dem Knoten* und *Geteilt* hängt davon ab,
wie gross die externe Tabelle ist, und wie viele Messungen dadurch nicht
an den externen Rechner geschickt werden müssen.

Übersicht

1. Motivation
2. Queryabarbeitung
3. Data Centric Storage
4. Tiny Aggregation

Speichern

Historische Abfragen

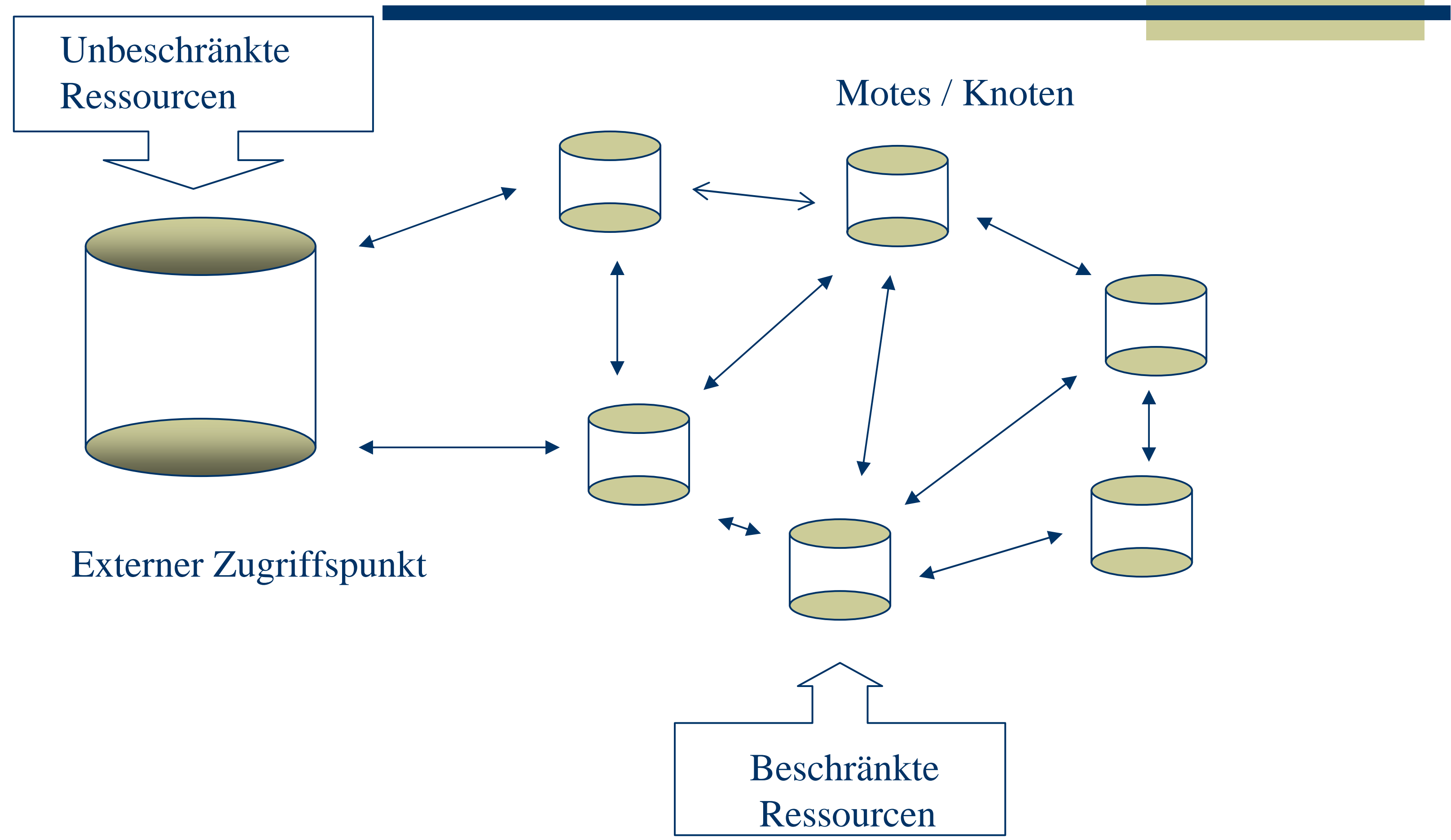
Für das Jahr 2002 alle Punkte der Schweiz die eine Durchschnittstemperatur von mehr als 24°C hatten.

```
SELECT AVG(s.temp), s.id
FROM Sensor s
WHERE s.generated > 1-1-2002 AND s.generated < 31-12-2002
GROUP BY s.id
```

Voraussetzungen

-Jeder Knoten verfügt über ein Ortungssystem (GPS)
-Sensornetz mit n Knoten und $\sqrt{n}$ Durchmesser
- Für Punkt zu Punkt Kommunikation braucht man höchstens $\sqrt{n}$ Schritte
- Zugriffspunkte sind stabil

Aufbau eines Sensornetzes



Speicherarten

Lokale Speicherung
Jeder Event wird dort gespeichert, wo er erzeugt wurde.
Speichern: O(0)
Query: O(n)

Externe Speicherung
Jeder Event, der auftritt wird an den externen Rechner
Speichern: $O(\sqrt{n})$
Query: O(0)

Data Centric Storage

f(Name des Events) = (x,y)

Events werden nun an diesem Ort gespeichert und gesucht.
Speichern: $O(\sqrt{n})$
Query: $O(\sqrt{n})$

Wie funktioniert Data Centric Storage?
Data Centric Storage ist eine Hashtabelle mit den beiden Operationen
void Put(key, value)
value Get(key)

Setzt sich aus GPSR und darauf aufbauend eine verteilte Hashtabelle

GPSR

In GPSR (Greedy Perimeter Stateless Routing) schicken Knoten Pakete nicht an einen expliziten Knoten, sondern an einen Ort. Empfangender Knoten ist der Nächste zu diesem Ort.

Voraussetzungen:

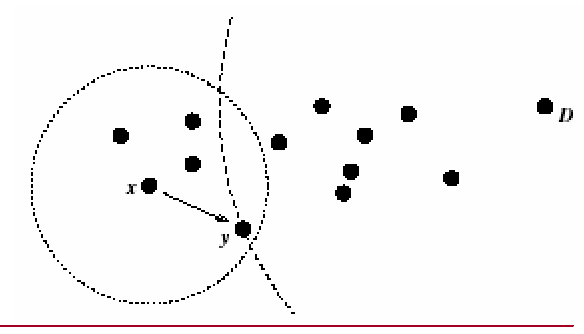
1. Jeder Knoten kennt seinen geographischen Ort, und aktualisiert diesen regelmässig.
2. Jeder Knoten kennt die Knoten, die in Reichweite seiner Funkantenne liegen, samt deren Positionen.

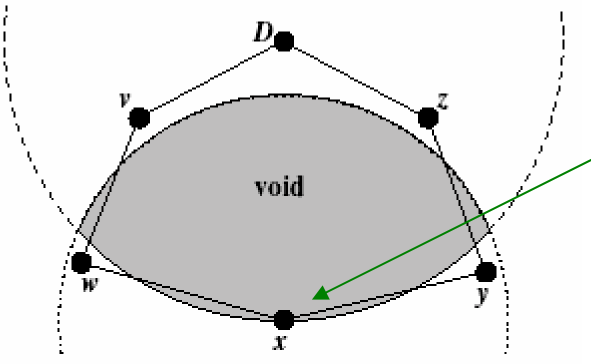
Id: 371
Position:
x=600’000, y=200‘000
Nachbarn:
x=600‘002, y=200‘003
x=599‘949, y=200‘001

GPSR

Greedy Forwarding:

In diesem Modus werden Pakete von einem Knoten x an den Knoten gesendet, die der Zieldestination am nächsten ist.



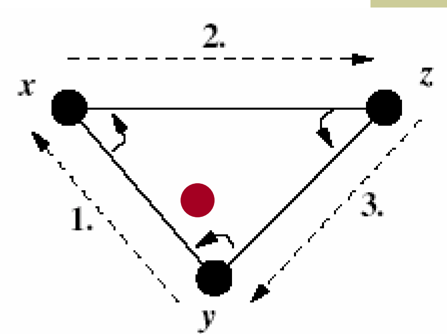


Dead end

x hat keinen Knoten in seiner Reichweite, der näher bei D ist, obwohl er nicht der Zielknoten ist.

GPSR

Perimeter Mode
Umkreisen von Löchern im Uhrzeigersinn, bis ein näherer Knoten erreicht wurde, oder zurück am Startpunkt ist.

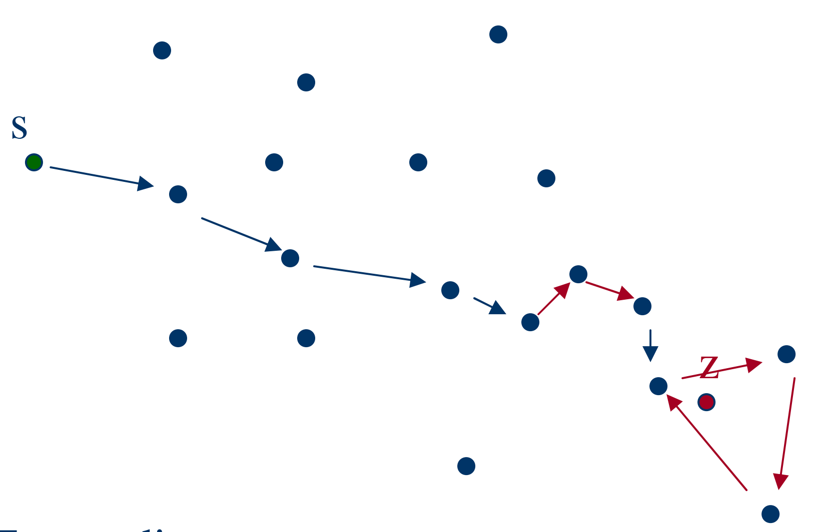


Paket
destination = {17/19}
perimeterMode = true
currentNearest = {8/11}

Wenn ein Knoten näher an dem Ziel ist, dann fällt er zurück in den *Greedy Forwarding Mode*, wenn er in seiner Nachbarschaft einen Knoten hat, der dem Ziel näher ist.
Sonst weiter in Perimeter Mode.

Ist zu Ende, wenn eine Umkreisung im Perimeter Mode gemacht wurde.

GPSR



Greedy Forwarding

Perimeter Mode

Distributed Hash Table (DHT) over GPSR

f(Name eines Events) = (x,y)

Hashfunktion f muss Grenzen des Sensornetzes in etwa kennen.

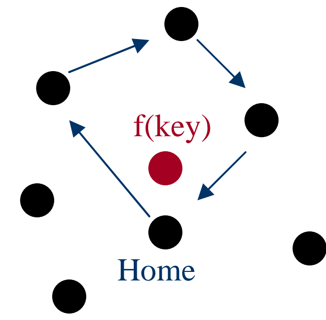
put(key, value) schickt value mit GPSR an f(key).
get(key) schickt eigene Koordinate x mit GPSR an f(key), danach wird mit GPSR value an x zurückgeschickt.

Bis jetzt ist Data Centric Storage nur für stationäre, ausfallsichere Netze zu gebrauchen.

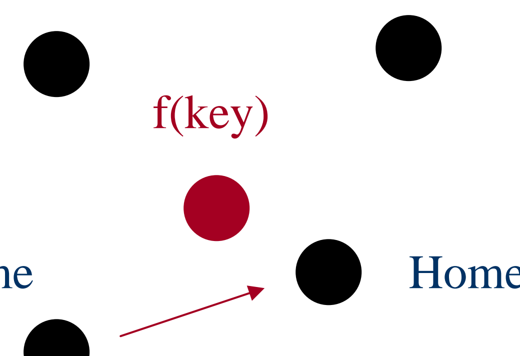
Robustness Extension

Sensornetze sind weder statisch noch ausfallsicher:

Folgender Algorithmus erweitert Data Centric Storage
Der Knoten, der der Koordinate f(key) am nächsten ist, wird *home* Knote für key genannt.
Dieser Knote wendet periodisch den GPSR Algorithmus an.



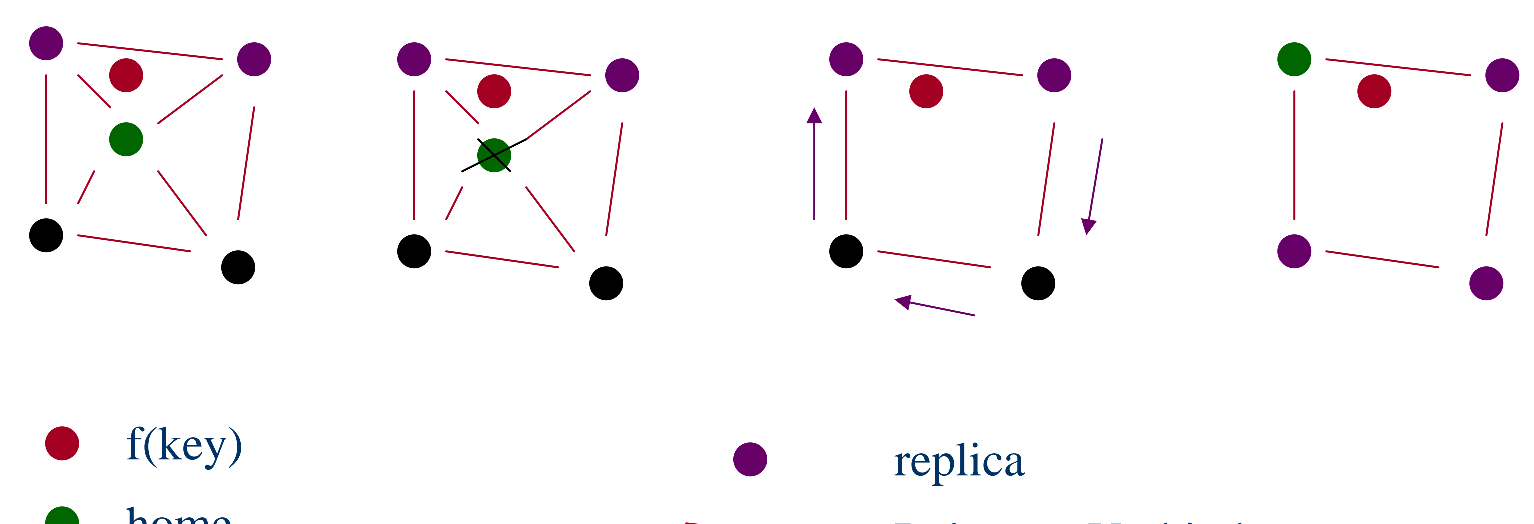
Eine solche Refresh Nachricht wird von einem näheren Knoten nicht weitergeleitet.
Näherer Knoten wird *home* Knoten.



Robustness Extension

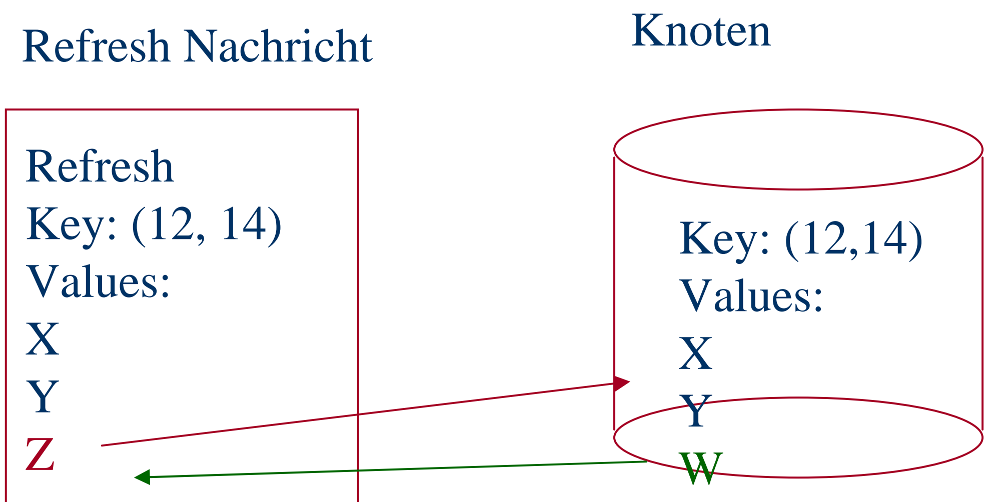
Ausfälle von Knoten

Bei Refresh-Nachrichten werden die Daten von allen Knoten zwischen – gespeichert, die diese erhalten. Wenn der *home* Knoten ausfällt, beginnt ein Knoten mit einer Kopie eine Refresh-Nachricht zu senden.



Robustness Extension

Bei inkonsistenten Daten, wird Delta hinzuaddiert.



Was wenn Knoten lokal häufiger ausfallen?

-Nicht nur ein f, sondern mehrere.

Kosten

Methode:	extern	lokal	Data centric
Kosten zum Speichern	**$O(\sqrt{n})$**	**0**	**$O(\sqrt{n})$**
Kosten um Query zu verschicken	**0**	**$O(n)$**	**$O(\sqrt{n})$**
Kosten um Resultat zurückzuschicken	**0**	**$O(\sqrt{n})$**	**$O(\sqrt{n})$**

Totalkosten setzten sich wie folgt zusammen:

D_{total}: Die totale Anzahl von Events die in einem Netzt erkannt werden.

Q: Die Anzahl von Querys die gestellt wird.

D_q: Die Menge der Antworten, die durch Q generiert wird

$Q << D_{total}$

Kosten

Externe Speicherung:

Total: $D_{total} * \sqrt{n}$

Hotspot: D_{total}

Lokale Speicherung

Total: $Q*n + D_q*\sqrt{n}$

Hotspot: $Q + D_q$

Data-Centric Speicherung

Total: $Q*\sqrt{n} + D_{total}*\sqrt{n} + D_q*\sqrt{n}$

Hotspot: $Q + D_q$

Scalability Extension

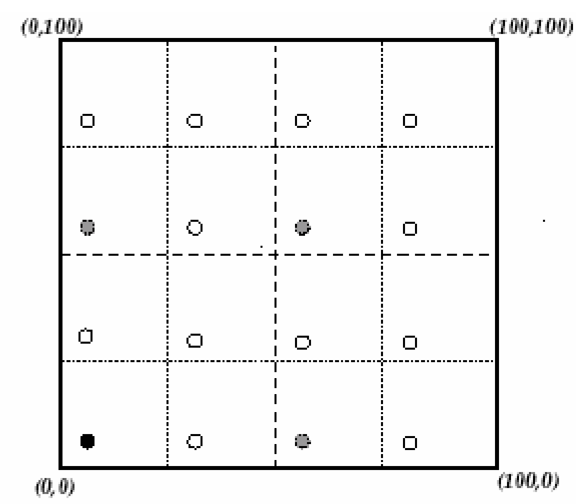
Bis jetzt ist externe Speicherung für totale Anzahl der Nachrichten am Besten.
Dafür ist Data Centric Storage beim Hotspot am Besten.

Da $Q << D_{total}$ versuchen Kosten pro D_{total} zu verkleinern.

f(key) ist root point.
Zusätzlich hat es 4^d mirror points.

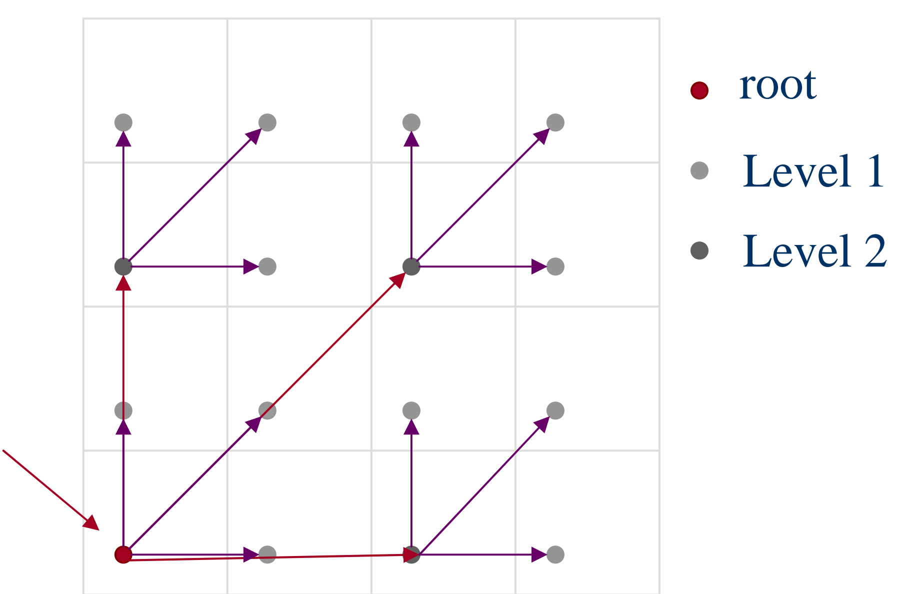
Speichern:
Ein Event wird am nächsten root oder mirror Point abgespeichert.

Kosten:
Von $O(\sqrt{n})$ nach $O(\sqrt{n} / 2^d)$



Scalability Extension

Query:
Eine Query erfolgt über den root point.
Von dort wird die Query an alle Mirror Points Level 1 geschickt, dann Level 2....
Value wird an root Punkt zurück geschickt.



Kosten:
$O(2^d * \sqrt{n})$

Weiterer Vorteil:
Hotspot Gefahr verkleinert

Kosten

Externe Speicherung:

Total: $D_{totaln}0.5$

Hotspot: D_{totaln}

Lokale Speicherung

Total: $Q*n + D_q*\sqrt{n}$

Hotspot: $Q + D_q$

Data-Centric Speicherung

Total: $Q*\sqrt{n} + D_{total}*\sqrt{n} + D_q*\sqrt{n}$

Hotspot: $Q + D_q$

Data-Centric Speicherung mit Scability Extension

Total: $Q* 2^d * \sqrt{n} + D_q * \sqrt{n} / 2^d$

Hotspot: $Q + D_q$

Test

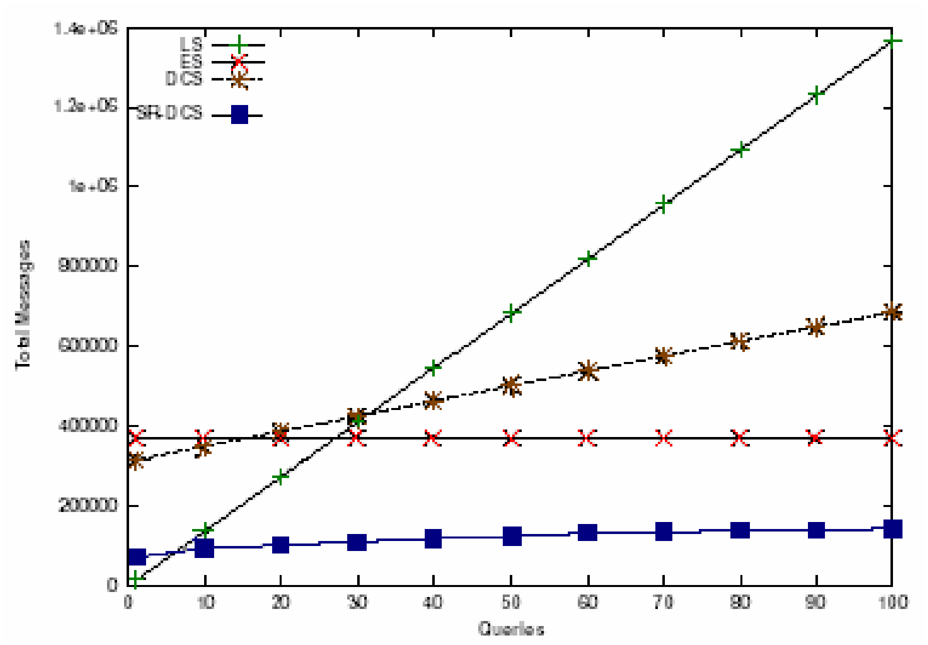
Testszenario:

n = 10'000 (Anzahl Knoten)

T = 100 Nummer von Eventtypen

D_i = 100 Die Anzahl der Events, die für Eventtyp i abgespeichert werden.

Q = Konstante



Übersicht

1. Motivation
2. Queryabarbeitung
3. Data Centric Storage
4. Tiny Aggregation

Aggregation

Was ist Aggregation

Aggregationen machen aus einem Set von Daten einen einzigen Wert.
SQL-99 hat folgende Aggregationen: SUM, MIN, MAX, COUNT, AVG.
Kommen häufig mit der GROUP BY Klausel vor
Bsp.:

Firma:

Mitarbeiter	Chef
Roli	Meyer
Fredi	Müller
Ruädi	Meyer
Hansli	Meyer

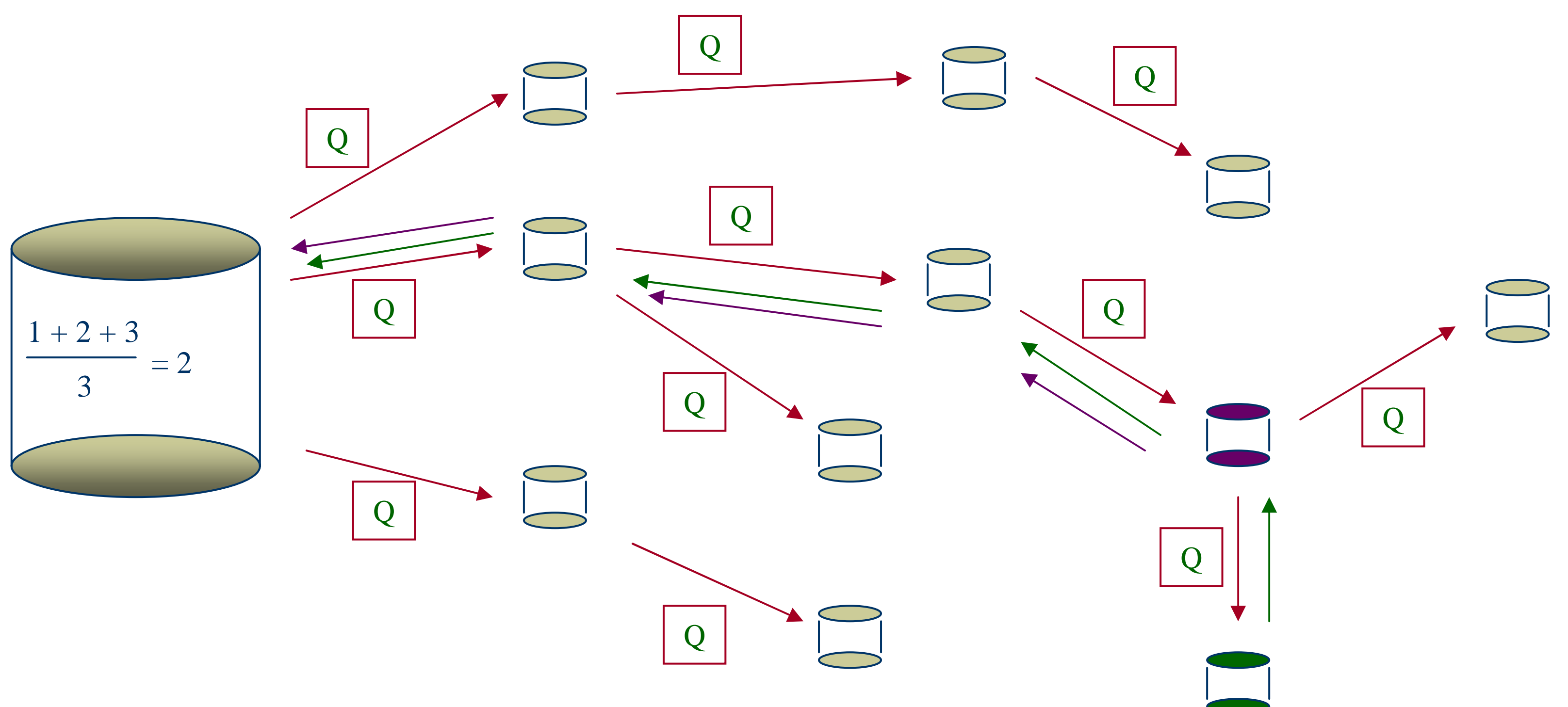
```
SELECT Chef, COUNT(Mitarbeiter)
FROM Firma
GROUP BY Chef
```

Chef	Count(Mitarbeiter)
Meyer	3
Müller	1

Aggregationen kommen häufig vor

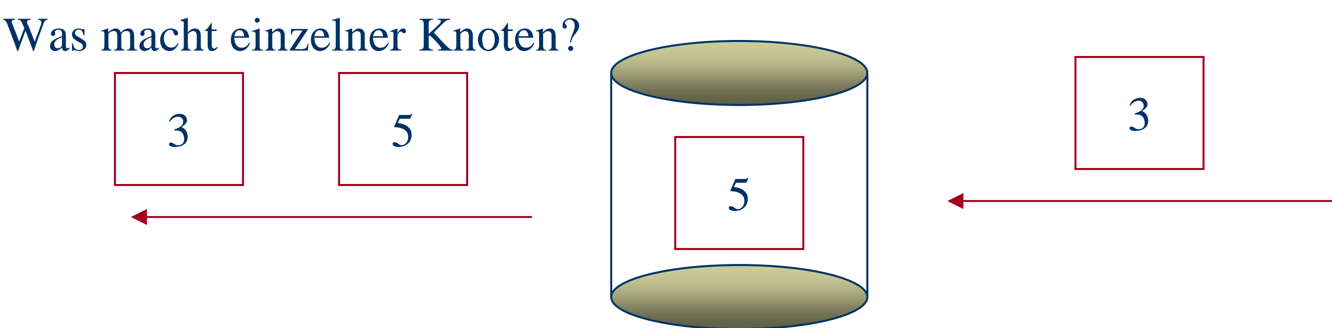
Ein einfacher Ansatz

Externer Rechner flutet Query ins Netz
Jeder Knoten schickt seinen Messwert dem externen Rechner

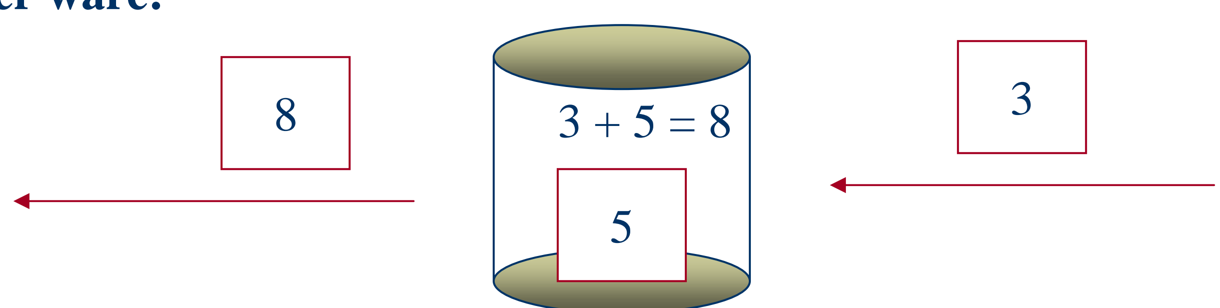


Ein einfacher Ansatz

Was macht einzelner Knoten?



Besser wäre:



Tiny Aggregation

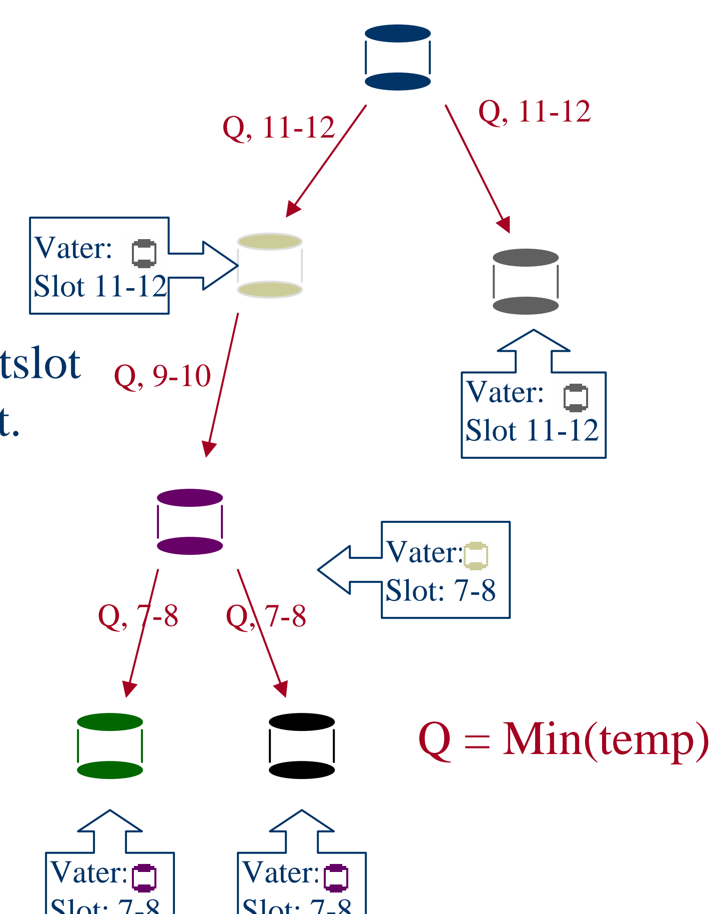
Dieser Algorithmus hat 2 Phasen:

Distribution Phase:

Query wird ins Sensornetz geflutet.

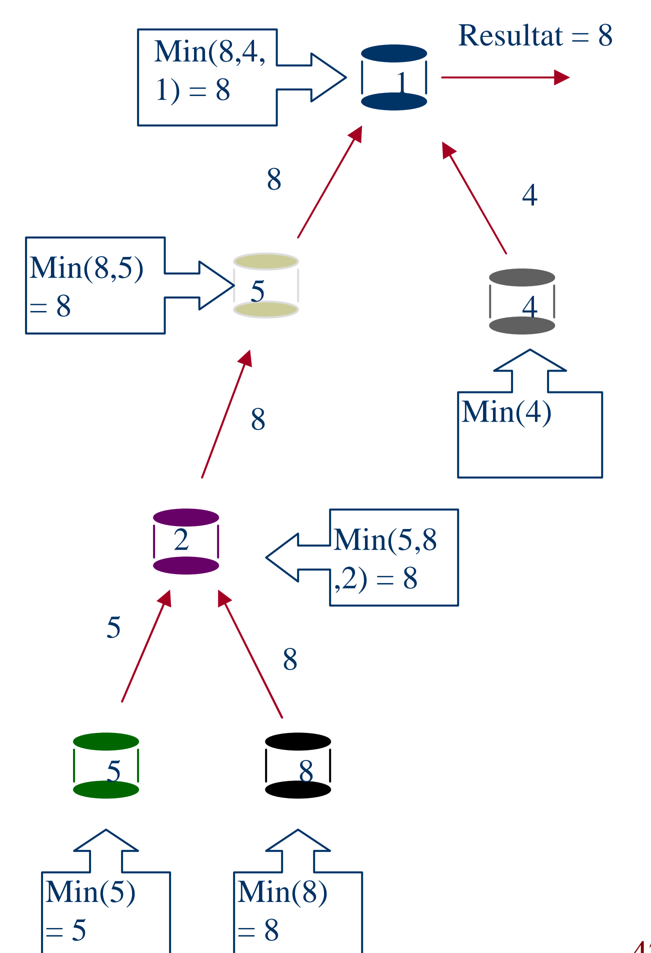
Jeder Knoten bestimmt den Sender der Query als Vater.

Jeder Vater erwartet in einem bestimmten Zeitslot eine Antwort und teilt dies seinen Kindern mit.

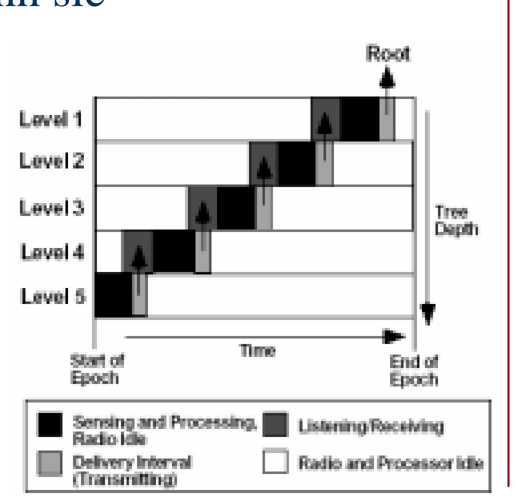


Tiny Aggregation

Collection Phase
Knoten warten auf die Antwort Ihrer Kinder.
Wenn Zeitslot vorbei, wird das lokale Aggregat berechnet aus den Werten der Kinder und dem eigenen Wert.



Knoten wissen, wann sie wieder Arbeit haben => Können in der Wartezeit schlafen

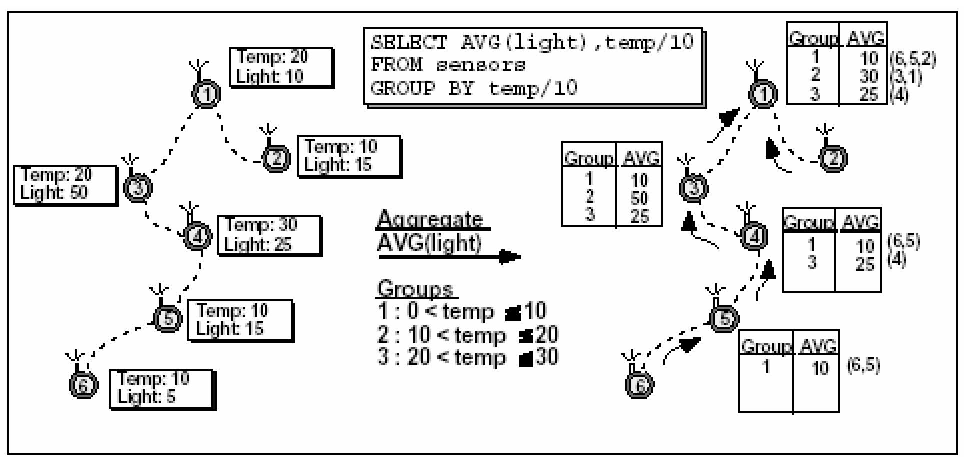


Grouping

Häufig kommen Aggregationen mit Grouping vor

SELECT AVG(light), temp/10
FROM sensors
GROUP BY temp / 10

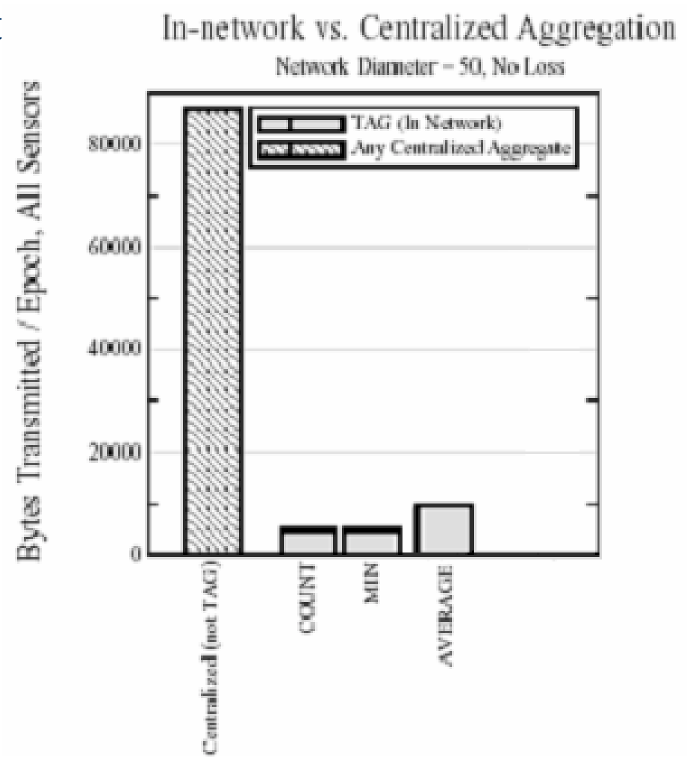
Es wird für jede Gruppe ein Aggregat gesendet.



Tests

Vorteile:
-Nachrichten einsparen
-Jeder Knoten sendet gleich viele Nachricht

Testsimulation mit 2‘500 Knoten
Knoten vernünftig verteilt.



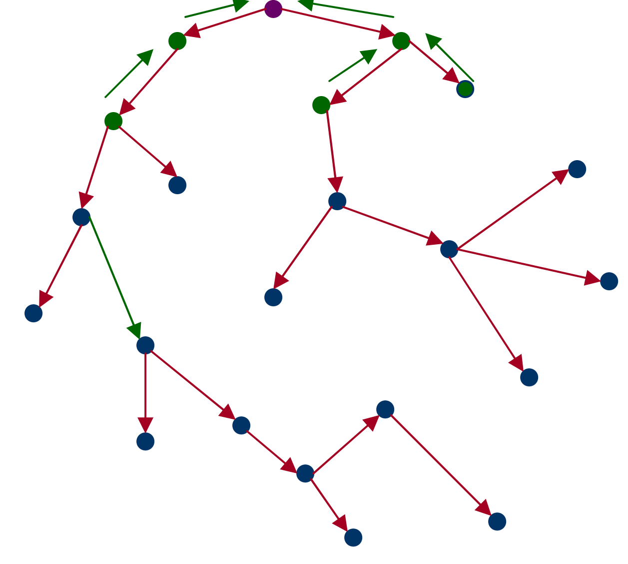
Optimierungen

Zuerst nur über die ersten d Ebenen des Baumes aggregieren

Neue Aggregation mit dem Resultat der ersten Aggregation.
Knote antwortet nur, wenn sein Teilaggregat etwas an der Lösung ändert.

Funktioniert für MIN und MAX, wenn Werte zufällig verteilt sind (!)

Bei AVG auch anwendbar: Fehlerschranke



Schlussbetrachtung

Bis jetzt viel Theorie, wenig implementiert.
Alle Ansätze versuchen Anzahl Nachrichten zu reduzieren.

Die endgültige Lösung wird eine Kombination von allen drei Ansätzen sein.

FRAGEN

?