Seminar Verteilte Systeme – Thema Sensornetze

Titel: Speicher und Datenbanktechniken
Name: Patrick Leuthold
Datum: 4.7.2003

Betreut von: Michael Rohs

Abstract:
In Sensornetzen werden Unmengen von Daten generiert. Datenbanktechniken erlauben es dem Nutzer, diejenigen zu selektieren, die er haben will. Dabei gilt auch hier, dass zur Energieeinsparung die Anzahl der gesendeten Nachrichten minimiert werden muss. In meinem Vortrage werden dazu drei Ansätze vorgestellt.
Erstens wird gezeigt, wie eine Query in Sensornetzen am besten abgearbeitet wird, zweitens wird eine Speichermethode vorgestellt, die Daten innerhalb der (dynamischen) Sensornetze speichert und dabei energieeffizienter ist als eine externe oder lokale Speicherung. Die dritte Betrachtung gilt der Aggregation der Messwerte, bei welcher dank einer frühzeitigen Zusammenfassung die Anzahl der Nachrichten minimiert werden kann.

1. Einführung & Motivation

1.1. Das Sensornetzwerk als Datenbank

Es ist einfach, ein Sensornetz als Datenbank zu betrachten. Während wir in relationalen Datenbanken Querys über Tabellen stellen, werden in einem Sensornetz Querys über Sensoren gestellt. Die Daten, die wir hierbei abfragen entsprechen der physikalischen Welt. Wir stellen eigentlich Querys auf die Umwelt der Sensoren. Das Sensornetz ist uns schlussendlich nur dann nütze, wenn es uns die Daten in vernünftiger Zeit liefert, die wir gebrauchen. Ein Sensornetz, das wir in diesem Bericht betrachten wollen, besteht aus den einzelnen Sensoren, sowie aus einem Access – Point, welcher sich im Gegensatz zu den einzelnen Knoten nicht mit Ressourcen-Problemen behaftet ist. (Stromversorgung übers Netz, grosse Speicher- und Rechenkapazitäten etc.) (siehe Abbildung 1.1.)
Momentane Sensorknoten können ca 0.5 MB Daten speichern. Wenn man davon ausgeht, dass pro Messwert 50 Byte Speicherplatz gebraucht werden, können immerhin schon 100'000 Messwerte abgespeichert werden. [1]

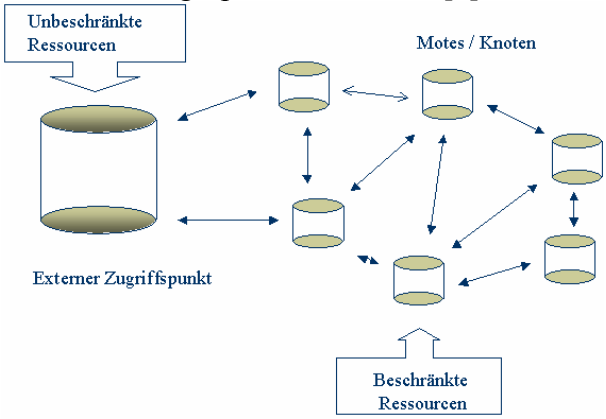


Abbildung 1.1.

1. 2. Verschiedene Suchabfragen

Ein Benutzer eines Sensornetzes kann diesem verschiede Arten von Suchabfragen stellen, die im folgenden kurz beschrieben werden.

1.2.1. Historische Abfragen

Wir wollen die Messwerte eines, oder mehrerer Sensoren, wissen:
Für das Jahr 2003, alle Punkte der Schweiz, die eine Durchschnittstemperatur von mehr als 24°C hatten.

1.2.2. Snapshot

Bei dieser Art von Querys ist man an den Messwerten von bestimmten Sensoren zum jetzigen Zeitpunkt interessiert.
Welche Durchschnittliche Temperatur messen die Sensoren in Zürich jetzt im Durchschnitt.

1.2.3. Laufende

Diese Querys liefern nach einem Startpunkt immer nach einem Zeitintervall die gewünschten Daten:
Für die nächste halbe Stunde möchte ich alle 30s die durchschnittliche Temperatur aller Sensoren in Zürich.

1. 3. SQL Querys in Sensornetzen

Die in 1.1. gezeigten möglichen Suchabfragen können mit der bereits bestehenden Querysprache SQL nicht gestellt werden. Um laufende Querys stellen zu können, muss man SQL um Zeitkonstrukte erweitern [2&3]:

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

2. Queryabarbeitung [2]

2.1. Die verschiedenen Strategien

Wenn wir in verteilten Sensornetzen mit Speicherfähigkeit Querys stellen, haben wir verschiedene Möglichkeiten dies zu tun. Nehmen wir folgende Suchabfrage:

```
SELECT VR.value, R.x, R.y
FROM RFSensor R, VRFSensorsGetRainfallLevel VR
WHERE R.Sensor.getTemperature() > 25 AND VR.Sensor = R.Sensor
AND EVERY(30)
```

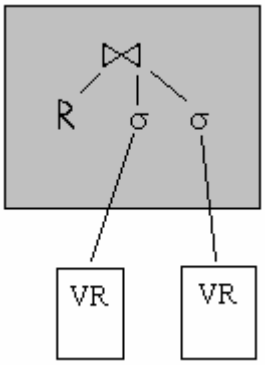


Abbildung 2.1.

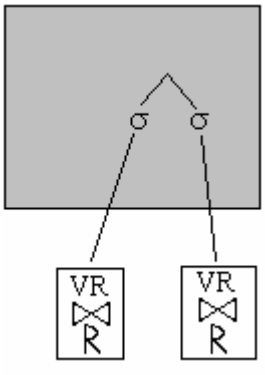


Abbildung 2.2

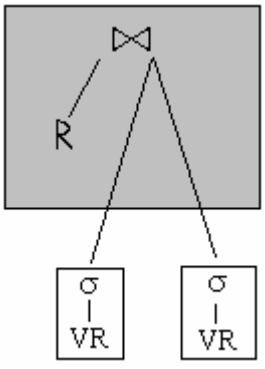


Abbildung 2.3.

VR ist dabei eine virtuelle Tabelle, die Messwerte der einzelnen Sensoren zu einer relationalen Tabelle zusammenfasst. R ist eine relationale Tabelle, die Zusatzinformationen zu den einzelnen Sensoren bereithält, in diesem Fall den Ort der Sensoren. Es mag auf den ersten Blick erstaunlich wirken, dass der Ort eines Sensors ausserhalb des Sensors abgespeichert wird. Wir könnten uns aber vorstellen, dass es eine Vielzahl von Tabellen geben könnte, die Zusatzinformationen zu den einzelnen Sensoren bereitstellen könnten, welche die Speicherkapazität eines Sensors sprengen würden.

Kommen wir zurück auf unsere verschiedenen Möglichkeiten diese Query auszuführen.
Die erste Möglichkeit ist, dass die Sensoren ihren Wert sammeln, und dem externen Rechner senden. Dort werden die Daten gemäss der WHERE - Condition selektioniert und anschliessend mit der Tabelle R verbunden, wie dies in Abbildung 2.1. dargestellt wird. Dies ist sehr unbefriedigend, da alle Messwerte von den Knoten an den externen Rechner gesendet werden, mit Messwerten, die eigentlich gar nicht gebraucht werden (alle Messwerte, die kleiner als 25 sind).
Abbildung 2.2. zeigt eine andere Möglichkeit: Die Attribute der Tabelle R, die für den Join notwendig sind, werden an die einzelnen Knoten gesendet. Dort werden die Operationen durchgeführt und die Resultatwerte werden an den externen Frontendserver gesendet.
Abbildung 2.3. zeigt die dritte Möglichkeit, die hier betrachtet wird. Es wird nur das Select Statement an die einzelnen Knoten gesendet, und danach das Resultat an den Frontendserver zurückgesendet. Dort wird die Join Operation mit der Tabelle R durchgeführt.

2.2. Tests

Um die drei verschieden Varianten zu vergleichen, wurde ein Test simuliert. Um die Kosten zu vergleichen, wurde folgende Formel verwendet:
*Kosten in Joules = Wcpu * CPU + Wram * RAM + Wsend * #Msgs + Wbdw * SizeMsgs*

Wobei die Werte der Parameter in Abbildung 2.4. dargestellt sind.

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

Abbildung 2.4.

Bei Wsend handelt es sich um die Synchronisationskosten zwischen Sender und Empfänger. Es wurde angenommen, dass jeder Knoten direkt mit dem externen Rechner verbunden ist. Es wurden zwei Extrema gewählt. Im ersten, dass keine Messung eine Temperatur über 25° Grad ist, und im zweiten Testfall, dass jede Messung über 25°C ist. Die Resultate sind in Abbildung 2.5. und 2.6. dargestellt.

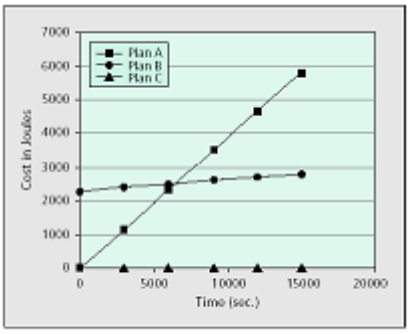


Abbildung 2.5.

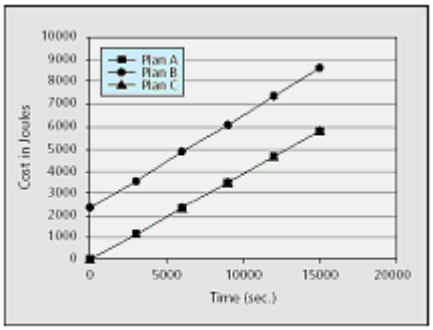


Abbildung 2.6.

Dabei wurde angenommen, dass alle Knoten direkt mit dem Frontendserver verbunden sind. Insgesamt wurde die Simulation mit 200 Knoten durchgeführt. Die Variante 2 (Plan B) schliesst deshalb so schlecht ab, da durch den Join auf dem Knoten kein Messwert weniger gesendet werden musste: Jedes Tupel in einer virtuellen Relation hat ein Gegenstück in R.

3. Data Centric Storage [3]

3.1. Motivation (Was ist Data Centric Storage?)

Wie der Name bereits sagt, ist bei Data Centric Speicherung nicht der Name eines Knotens im Mittelpunkt, sondern die Daten selber.
Nicht der Name des Knotens ist wichtig, sondern die Namen der Daten die dieser Knoten an einem bestimmten Ort gesammelt hat, sind wichtig.
Die Daten müssen deshalb entsprechend „adressiert" werden, damit sie im Netzwerk, unabhängig auf welchem Knoten sie sich befinden, gefunden werden können.
Es werden drei verschieden Speichermöglichkeiten in Sensornetzen untersucht. Es wird dabei ein Sensornetz mit n Knoten betrachtet, wobei der Durchmesser $\sqrt{n}$ beträgt. Wenn eine Nachricht von einem Knoten des Sensornetzes an einen anderen gesendet wird, müssen $\sqrt{n}$ Nachrichten gesendet werden, wenn der Aufenthaltsort bekannt ist. Wenn eine Nachricht an einen Knoten gesendet wird, dessen Aufenthaltsort nicht bekannt ist, muss das Netz geflutet und O(n) Nachrichten versendet werden.

3.1.1. Externe Speicherung(ES)

Sobald ein Event (unter Event werden in diesem Zusammenhang Daten genannt, die abgespeichert werden) abzuspeichern ist, wird er an einen externen Speicher gesendet, und dort für spätere Verwendung abgespeichert. Für jeden aufgetreten Event ergeben sich Kosten von $O(\sqrt{n})$. Dafür gibt es keine Kommunikationskosten, wenn wir auf diesen Daten Querys ausführen.

3.1.2. Local Storage(LS)

Die Daten werden lokal dort gespeichert, wo sie gemessen wurden. Wir haben keine Kommunikationskosten, wenn die Daten gemessen wurden, dafür Kosten von O(n), wenn wir eine Query zu dem entsprechenden Knoten fluten, und Kosten von $O(\sqrt{n})$ um das Resultat zurück an die Wurzel zu schicken.

3.1.3. Data-Centric Storage

Hier wird nach dem Auftreten eines Events, dieser an den richtigen Ort gesendet. Dabei entstehen Kosten von $O(\sqrt{n})$. Weil der Ort der Daten bekannt ist, können Querys mit Kosten $O(\sqrt{n})$ an den Ort der Daten geschickt werden. Die Daten zurückzusenden, kostet dabei wieder $O(\sqrt{n})$.

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

Des Weiteren werden noch die totalen Kosten des ganzen Netzes betrachten, die für die jeweilige Methode gebraucht werden. Zusätzlich werfe ich noch einen Blick auf die Anzahl Pakete, die der Knoten verarbeiten muss, der am meisten beansprucht wird. Am meisten beansprucht in einem Sensornetz wird der Knoten, der dem externen Zugriffspunkt am nächsten ist. Dies gilt, da jede Query Nachricht und jede Nachricht mit dem Resultat diesen Punkt passieren muss.

Es gibt dabei folgende Grössen, die bei den nachfolgenden Berechnungen gebraucht werden:

Dtotal: Die totale Anzahl von Events, die in einem Netz erkannt werden.
Q: Die Anzahl von Querys die gestellt werden.
Dq: Die Menge der Antworten, die durch Q generiert werden.

Externe Speicherung:

Total: $D_{total} * O(\sqrt{n})$

Hotspot: D_{total}

Lokale Speicherung

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

Hotspot: $Q + D_q$

Data-Centric Speicherung

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

Hotspot: $Q + D_q$

Bei diesem ersten Kostenvergleich ist überraschender Weise die Externe Methode die beste, dies könnte sich aber noch ändern, wenn Data-Centric Speicherung verbessert wird.

3.3. Wie funktioniert Data Centric Storage

Data Centric Storage entspricht einer Hash-Tabelle, welche die folgenden beiden Basisfunktionen beinhaltet:
Put(key, value)
Get(key)
Diese beiden Funktionen werden auf zwei Arten unterstützt. Zuerst wird der GPSR (Greedy Perimeter Stateless Routing) Algorithmus für ein Low-Level Routing verwendet. Danach wird ein effizienter Peer-to-Peer Suchalgorithmus angewendet, um eine Verteilte Hash-Tabelle darauf aufzubauen. In den beiden folgenden Abschnitten werden diese beiden Algorithmen genauer erklärt.

Abbildung 3.1.

3.3.1. GPSR (Greedy Perimeter Stateless Routing)

In diesem Algorithmus muss folgende Voraussetzung gelten: Jeder Knoten kennt seinen Ort, beispielsweise dank GPS. Er überprüft seinen Ort periodisch, und teilt auch periodisch seinen Ort seinen Nachbarn (die sich in seinem Funkradius befinden) mit.
Um ein Paket zu versenden, wird das Paket nicht mit einer Adresse adressiert, die den Zielknoten identifiziert, sondern es wird die Zielkoordinate in den Paketheader geschrieben. Der Zielknoten ist der, der dieser Koordinate am nächsten ist. Ein Paket wird bei diesem Algorithmus immer an den Knoten gesendet, der der Ziel - Koordinate am nächsten ist (siehe Abbildung 3.1). Wenn ein Paket einen Knoten erreicht, der keinen Nachbarn hat, der näher an dem Ziel ist als er selber, haben wir ein sogenanntes Dead-End erreicht. Nun kann es sein, dass dieser Knoten, der Zielknoten ist, es kann aber auch sein, dass eine Situation wie in Abbildung 3.2. dargestellt ist, eingetreten ist: x hat keinen Knoten in der Reichweite seines Funksignals, der näher zu D ist, als er selbst, wobei die gepunktete Linie die Reichweite

seines Radio Signals darstellt. Es gibt jedoch v und z die näher zu D sind, wie anhand der gestrichelten Linie erkannt werden kann. Über w könnte das Paket den Knoten v erreichen, und über y den Knoten z, welche beide näher bei D sind.
GPSR geht nun wie folgt vor: Solange ein Paket an einen Knoten, der näher am Ziel ist, gesendet werden kann, wird es an einen solchen gesendet. Sobald wir einen Knoten erreichen, der keinen Knoten mehr in der Nachbarschaft hat, der näher zum Ziel ist, wird im Header des Pakets ein Flag gesetzt, und im Uhrzeigersinn um die Fläche geschickt (Abbildung 3.3.), die in Abbildung 3.2. als *void* bezeichnet wird. In diesem Modus bleibt das Paket, bis es einen Knoten erreicht, der näher ist, als der im Paketheader angegebe nächste Knoten, oder es wieder zurück an den Knoten kommt, der das Paket in diesen Zustand gesetzt hat. Im ersten Fall wird gleich verfahren wie mit einem Paket, bevor man den Umkreisungszustand erreicht hat. Im zweiten Fall hat das Paket das Ziel erreicht. Dieser Algorithmus wird im folgenden Abschnitt gebraucht.

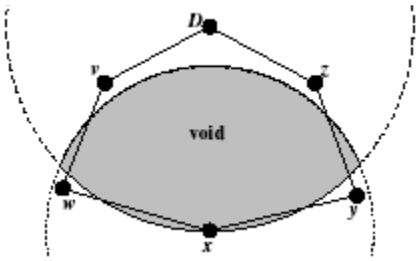


Abbildung 3.2.

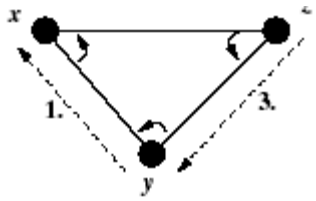


Abbildung 3.3.

3.3.2. Distributed Hash Table (DHT) over GPSR
Die Idee die hier gebraucht wird ist simpel: Gegeben sei ein Name eines Events. Dieser wird mit einer Hashfunktion auf eine Position innerhalb des Sensornetzes abgebildet. Wichtig ist, dass die Hashfunktion die Grenzen des Sensornetzes kennt, damit nicht ein Schlüssel generiert wird, der ausserhalb des Sensornetzes liegt. Die Funktion *put(key, value)* schickt das Paket mit den entsprechenden Daten (value) an den Ort, der durch den Schlüssel repräsentiert wird. Dafür wird der oben erklärte GPSR Algorithmus gebraucht, der das Paket an den Knoten schickt, der diesem Ort am nächsten liegt.
Auf gleiche Weise funktioniert die *get(key)* Operation. Sie wird mit dem GPSR Algorithmus an den nächsten Knoten von *key* geschickt, und der Knoten schickt die Daten an den Aufrufer dieser Funktion zurück.
Diese Lösung ist aber nur brauchbar, wenn wir ein stationäres Netz haben, und die Knoten ausfallsicher sind. Das nachfolgende Kapitel zeigt eine Erweiterung, damit Data – Centric Speicherung auch bei dynamischen Netzen funktioniert.

3.3.3 Robustness Extension
In einem realen Sensornetzwerk hat man zwei Probleme zu behandeln: dynamische und neue Knoten, sowie der Ausfall bestehender Knoten.
Folgender Algorithmus modifiziert den Data-Centric Speicher:
Jeder Event hat als *Home* Knoten den, der Koordinate am nächsten ist. Dieser sendet eine Refresh – Nachricht, welche alle Daten beinhaltet, die zu diesem Schlüssel gehören. (Gleich wie beim GPSR Algorithmus)
Wenn die Dynamik des Netzes dazu geführt hat, dass ein neuer Knoten der Schlüsselkoordinate am nächsten ist, dann wird dieser Knoten zum neuen Home – Knoten.
Um dem Ausfall von Knoten vorzubeugen, werden, wenn eine Refresh – Nachricht gesendet wird, die Daten zu dem entsprechenden Key von allen Knoten, die diese Refresh – Nachricht

erhalten haben gespeichert. Auch wenn nun mehrer Knoten die Daten gespeichert haben, sendet nur der *Home* Knoten die Refresh – Nachricht.
Damit beim Ausfall des *Home* Knotens ein neuer bestimmt werden kann, beginnen die Knoten, die die Daten ebenfalls gespeichert haben, Refresh – Nachrichten zu senden, bis ein neuer Home – Knoten gefunden wurde. Wenn ein Knoten aus einem solchen Kreis ausscheidet, muss er die Daten nicht mehr speichern. Deshalb löschen Knoten die Daten, wenn sie eine gewisse Zeit keine Refresh – Nachrichten mehr erhalten. Damit Inkosistenz zwischen den einzelnen Knoten verhindert werden kann, wird, wenn eine Refresh – Nachricht einen Knoten erreicht, die Refresh – Nachricht mit den Daten ergänzt, die ein Knoten zusätzlich noch hat.
Diese Erweiterung ist gut, wenn Knoten zufällig verteilt über das ganze Sensornetzgebiet ausfallen. In Sensornetzen kann es aber durchaus vorkommen, dass lokal viele Sensornetze auf einmal ausfallen, dann nützt die vorgestellte lokale Datenduplikation wenig. Es ist deshalb sinnvoll, ein Event an mehreren Orten gleichzeitig zu speichern, in dem man verschiede Hash – Funktionen anwendet.

3.3.4. Scability Extension

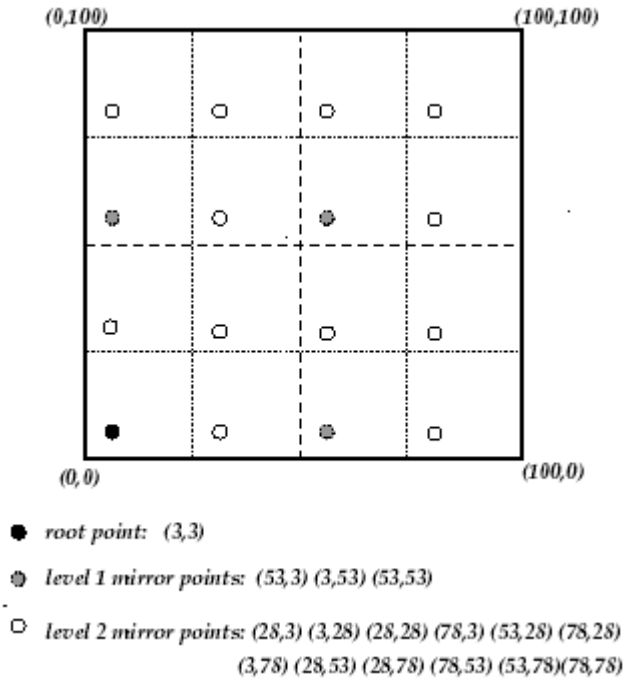


Abbildung 3.4.

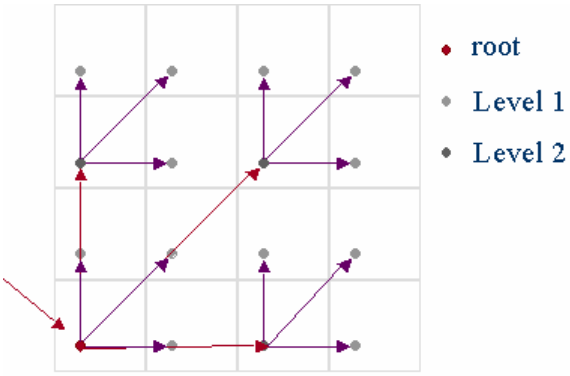


Abbildung 3.5.

Eine interessante Erweiterung wird in diesem Abschnitt vorgestellt. Es wird erlaubt, dass ein Event an mehreren Koordinaten abgespeichert werden kann, wie in Abbildung 3.4. abgebildet ist. Den Punkt den wir bis anhin als Event Punkt betrachtet haben, bildet dabei *den Root Point*. Wenn sich nun im Sensornetz ein Event ereignet, dann wird dieser nun nicht mehr zwingend an dem Ort gespeichert, wo der Event eigentlich hinzeigt, sondern es wird der Punkt der Replikas und des Event Punktes genommen, der dem Punkt am nächsten ist. Die Suche erfolgt über den Root Punkt und erfolgt dann wie in einem Baum, wie in Abbildung 3.5. dargestellt wird. Dadurch lassen sich die Kosten, die entstehen um einen Event

abzuspeichern, auf $O(\frac{\sqrt{n}}{2^d})$ senken. Die Kosten für eine Query steigen aber auf $O(2^d\sqrt{n})$.
Zudem können allfällige Hotspots, die entstehen, wenn sich von einem Queryevent mehrere gleichzeitig ereignen, vermindert werden.

3.4. Tests

Um die Qualität von Data Centric Storage darstellen zu können, wurden verschiedene Tests durchgeführt. Abbildung 3.6. zeigt wie sich die totale Anzahl der Querys in einem Sensornetz bei den verschiedenen Speichertechniken verhält. Es wurden dabei folgende Konstanten angenommen:
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
LS steht dabei für Lokalspeicherung, *ES* für Externe Speicherung.
D CS für Data Centric Storage und *SR D CS* für Scability Extension.

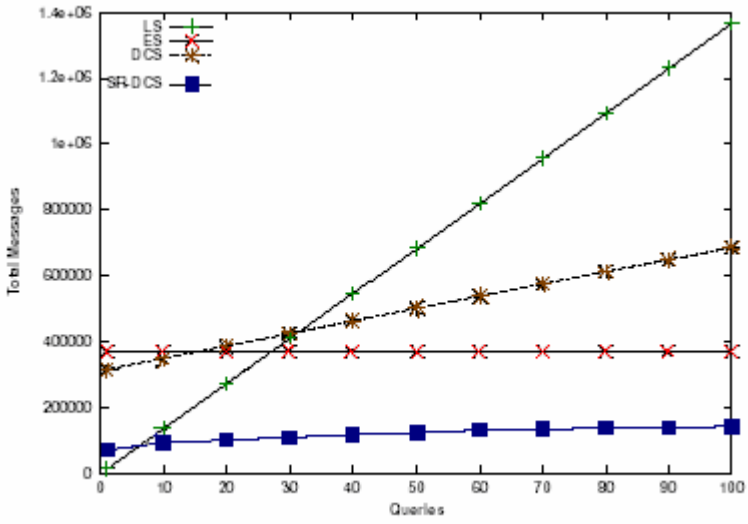


Abbildung 3.6.

Auch wenn in dieser Simulation angenommen wurde, dass die Knoten stabil sind, also für die Data Centric Methoden keine zusätzlichen Nachrichten gebraucht wurden, um die Daten am richtigen Ort zu halten, zeigt Data Centric Storage mit Scability Extension doch deutliche Vorteile gegenüber den anderen Speichermethoden.

4. TAG: a Tiny Aggregation Service for Ad-Hoc Sensor Networks [4]

In diesem Abschnitt wollen wir die Ansätze betrachten, die von der Universität von Berkeley gemacht wurden. Insbesondere legen wir in diesem Abschnitt einen Augenmerk auf die Aggregation.

4.1. Was ist Aggregation

Bei der Aggregation geht es darum, aus vielen Daten dem User eine zusammengefasste Version zu bieten. Typische Funktionen, die auch in SQL gebraucht werden, sind: SUM, MIN, MAX, COUNT, AVG

4.2. Wieso Aggregation

Eine einfache Möglichkeit, in Sensornetzen Aggregation zu betreiben, wäre, alle Daten, die generiert werden, an den externen Server zu senden und dort zu aggregieren. Dies ist aber nicht optimal, da unnötig viele Nachrichten gesendet werden. Besser wäre eine Variante, bei der möglichst viel im Netz zusammengefasst wird, und dadurch die Anzahl der gesendeten Nachrichten minimiert werden kann. Genau diesen Ansatz verfolgt die Tiny Aggregation (TAG).

4.3. Tiny Aggregation

Der hier vorgestellte Algorithmus besitzt zwei Phasen. In der ersten Phase wird die Query in das Netz hinein geflutet, die sogenannte *Distribution Phase*, und der zweite Phase, die *Collection Phase*, in welcher die Daten zurück an die Wurzel geroutet werden (Abbildung 4.1.).
Während der Distribution Phase sendet ein Knoten die Querynachricht an alle seine Nachbarn, von denen er die Querynachricht (Nachfolgend als Request bezeichnet) nicht

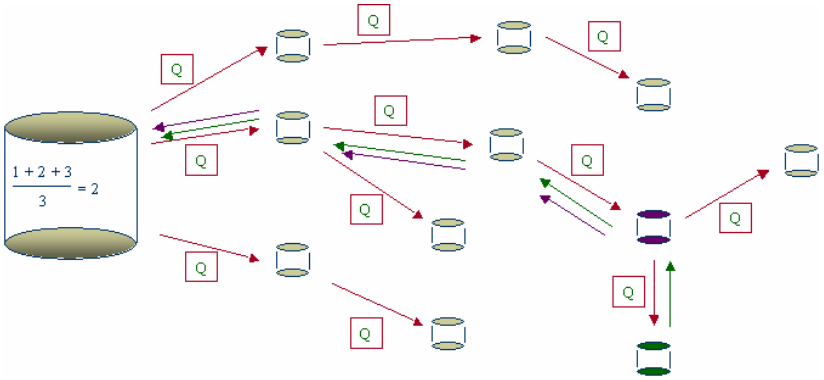


Abbildung 4.1.

erhalten hat. Die Knoten, die von ihm diese Nachricht als erste erhalten sind seine Kinder. Der Knoten sagt seinen Kindern, in welchem Zeitslot er eine Antwort von ihnen erwartet. Dieser Zeitslot wird so gewählt, dass der Knoten gerade noch Zeit hat, die Antworten zu aggregieren, und seinem Vater zu senden.

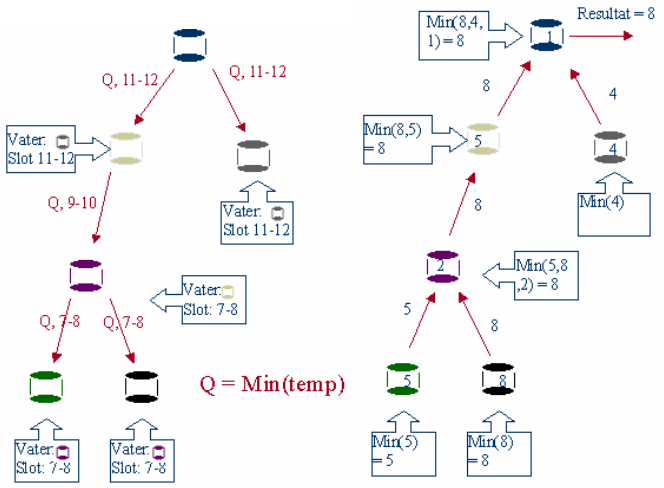


Abbildung 4.2.

Abbildung 4.3.

Nach dieser Phase wartet jeder Knoten auf die Antwort seiner Kinder. Nachdem der angegeben Zeitslot abgelaufen ist, berechnet ein Knoten aus den Antworten seiner Kinder und seinem eigenen Messergebnis, die Aggregation, und sendet diese seinem Vater. Anzumerken sei noch, dass die Vaterknoten etwas länger auf eine Antwort Ihrer Kinder warten, um allfällige Abweichungen in der Uhren-Synchronisation abzufangen. Dank diesem Algorithmus können die Knoten während einem grossen Teil der Zeit schlafen.
Im folgenden Abschnitten wird eine erweiterte Anwendungsmöglichkeit vorgestellt.

4.4. Grouping

Grouping in TAG entspricht dem GROUP BY in SQL. Wie die Tiny Aggregation hier angewendet wird möchte ich an folgendem Beispiel illustrieren. Gegeben sei folgende Query:

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

Man möchte zu jeder Temperatur-Zehnereinheit wissen, wie stark die durchschnittliche Sonnenbestrahlung ist. Anstatt nun jeder Knoten seinen Wert der Wurzel schickt, möchten wir TAG anwenden. Die Teillösung, die jeder Knote aggregiert, wird dabei ähnlich aggregiert, wie in Abschnitt 4.3. beschrieben. Mit der Ausnahme, dass ein Knoten nun nicht ein Resultat seinem Vater schickt (wie dies Beispielsweise bei einer einfachen MIN Aggregation der Fall wäre), nein, es wird für jede Gruppe ein Resultat aggregiert, sofern es bereits Messwerte für diese Gruppe gibt, wie dies in Abbildung 4.5. betrachtet werden kann.

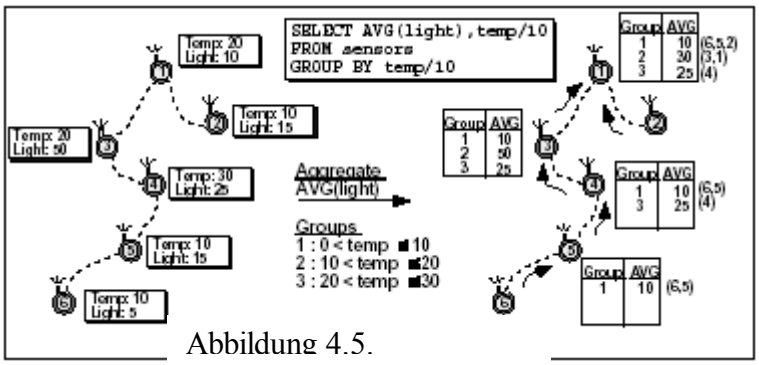


Abbildung 4.5.

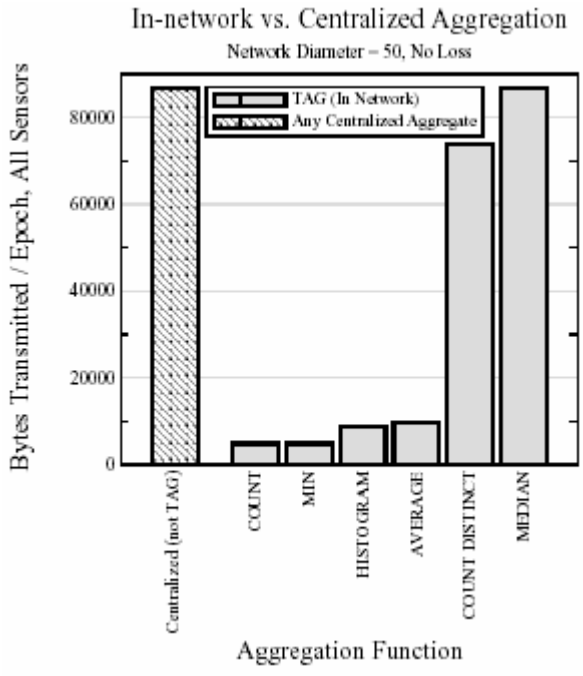
4.5. Vorteile von TAG

Tag hat gegenüber einer externen Aggregation vor allem den Vorteil, dass Nachrichten gespart werden können. Da ein Knoten festlegt, wann er eine Nachricht seiner Kinder erwartet, kann er die restliche Zeit im Schlafmodus verbringen. Ein weiterer Vorteil ist, dass

alle Knoten in etwa gleich viele Nachrichten senden, und deshalb auch etwa gleich viel Energie gebrauchen. Im Gegensatz zu einer externen Lösung, in der die Knoten, die sich nahe der Wurzel befinden, ein Vielfaches der Nachrichten senden müssen, wie ein Knoten der weite von der Wurzel entfernt ist.

4.6. Performance von TAG

Die UC Berkley hat auch Test über die Aggregation gemacht. Dabei wurde eine Simulation mit 2'500 Knoten durchgeführt, die regelmässig über das ganze Netz verteilt sind, da der Vorteil von TAG sehr stark davon abhängt, wie die Knoten über einem Feld verteilt sind. (Haben alle Knoten eine Verbindung zu der Wurzel, dann bringt TAG gar nichts, sind sie jedoch in einer Linie angereiht, haben wir den maximalen Nutzen.) Auch ist der Vorteil für die Verschiedenen Varianten von Aggregations – Operationen nicht immer gleich, wie sich aus Abbildung 4.7. unschwer erkennen lässt.



5. Referenzen

1. Yong Yao, Johannes Gehrke: Query Processing in Sensor Networks. 2003 CIDR Conference, http://www.cs.cornell.edu/johannes/papers/2003/cidr2003-sensor.pdf
2. Bonnet, Gehrke, Seshadri: Querying the Physical World, October 2000 issue of *IEEE Personal Communications* http://www.inf.ethz.ch/vs/edu/SS2003/DS/papers/storage/QueryingThePhysicalWorld.pdf
3. Ratnasamy, D. Estrin, R. Govindan, B. Karp, S. Shenker, L. Yin, and F. Yu, Data-Centric Storage in Sensornets Workshop on Sensor Networks and Application, Atlanta, GA, Sept. 2002. http://www.cs.virginia.edu/~son/cs862.s03/papers/data.storage.pdf
4. Madden, Franklin, Hellerstein, Hong: TAG: a Tiny AGgregation Service for Ad-Hoc Sensor Networks 5th Annual Symposium on Operating Systems Design and Implementation (OSDI). December, 2002. http://www.cs.berkeley.edu/~madden/madden_tag.pdf