Integriertes Seminar
*Datenbanken und Informationssysteme*

# P2P-Computing

**Lehrgebiet Datenverwaltungssysteme**
Prof. Dr. Dr. h.c. Härder
Prof. Dr. Deßloch

Björn Jung
b_jun@informatik.uni-kl.de



# P2P-Computing

Was sind Peer-to-Peer Systeme?

Wie kann man diese effizient nutzen?

Inwieweit ist es möglich Datenbanksysteme auf P2P Netzen aufzubauen?

# Gliederung

- Klassifikation von Netzstrukturen
- Ressource Discovery
- Strukturierte P2P-Architekturen
- Verarbeitung komplexer Queries in strukturierten P2P-Architekturen
- Zusammenfassung

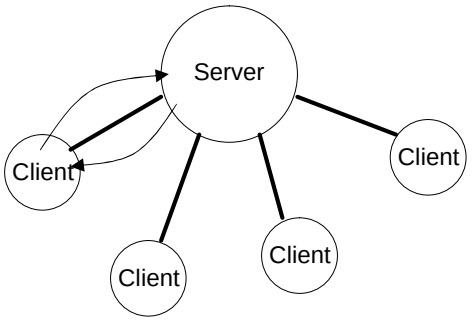


# Klassifikation von Netzstrukturen

- Server- / Clientsysteme
- hybride P2P-Systeme
- reine P2P-Systeme

# Klassifikation von Netzstrukturen

- Client- / Serversysteme




Server ist zentrale Anlaufstelle

Vorteile:
•einfach zu erstellen
•einfach zu warten

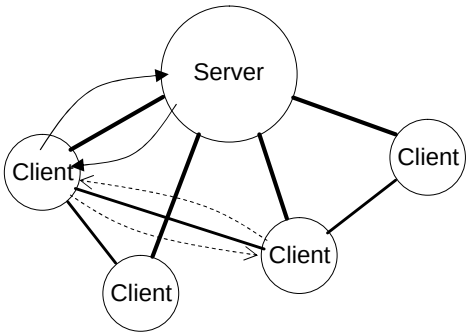
Nachteile:
•*Single-Point-of-Failure*
•nicht skalierbar

Bsp.: Ftp-Server



# Klassifikation von Netzstrukturen

- Hybride P2P-Systeme




Server ist zentrale Stelle

Kommunikation zwischen Clients möglich

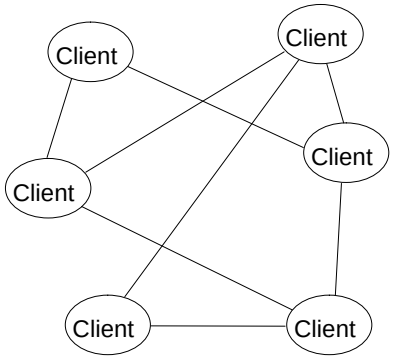
Vorteile:
•einfach zu erstellen
•einfach zu warten

Nachteile:
•*Single-Point-of-Failure*
•nicht skalierbar

Bsp.: Napster

# Klassifikation von Netzstrukturen

- Reine P2P-Systeme




keine Unterscheidung in Server und Client

Vorteile:
• skalierbar

Nachteile:
• *Einstiegsproblem*

jeder Knoten bietet eigene Ressourcen an

jeder Knoten kann auf die Ressourcen jedes anderen Knotens zugreifen

Bsp.: Gnutella

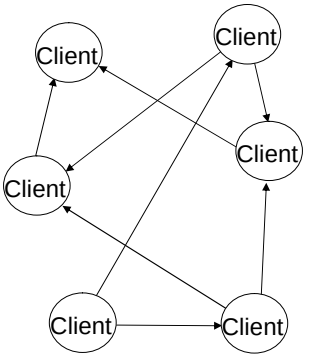


# Ressource Discovery

- Suche nach Objekten innerhalb des P2P-Netzes
  - **unstrukturierte Systeme**
    - basieren auf Broadcast
  - **strukturierte Systeme**
    - basieren auf Distributed-Hash-Tables

# RD - unstrukturierte Systeme

- jeder Knoten kennt nur seine eigene Daten
- jeder Knoten hat C Verbindungen zu anderen Knoten
- Suche per Broadcast im Netz




**Anzahl der Nachrichten:**

$$2 \sum_{i=0}^{hops} ( C(C-1)^i )$$



# RD - unstrukturierte Systeme

- Ansätze zur Reduzierung des Datenaufkommens:
  - **Time-To-Live (TTL) Wert**
    - jede Anfrage erhält einen Zähler der bei jedem hop dekrementiert wird
    - potentielle Daten werden nicht gefunden
  - **schrittweise Erhöhung des TTL Werts**

# RD - unstrukturierte Systeme

- Ansätze zur Reduzierung des Datenaufkommens:
  - **Unicast**
    - Anfrage nur noch an einen einzigen Knoten weiterreichen
    - hoher TTL Wert
    - Wahl des nächsten Knotens beliebig



# RD - unstrukturierte Systeme

- Ansätze zur Reduzierung des Datenaufkommens:
  - **dynamische Restrukturierung**
    - Annahme:
      Knoten die in der Vergangenheit gute Ergebnisse geliefert haben, werden dies auch in Zukunft
    - zu diesen Knoten direkte Verbindung herstellen
    - Clusterbildung
    - Überlastung von Knoten möglich

# RD - strukturierte Systeme

- Speicherort der Daten wird in eine Distributed-Hash-Table (DHT) eingetragen
  - **speichert (key,value)-Paare**
    - key: ist der gehashte Objektbezeichner
    - value: IP des Knotens, der das Objekt besitzt
  - **jeder Knoten für einen Teil der DHT zuständig**
  - **Knoten speichern nicht die (key, value)-Paare ihrer eigenen Daten**



# RD - strukturierte Systeme

- Funktionen der DHT
  - **put(key, value)**
  - **delete(key)**
  - **lookup(key) ? value**

Bsp: Tabelle mit 5-Bit keys

| Wert | value (IP) |
|---|---|
| 00011 (3) | 131.246.120.30 |
| 01001 (9) | 62.80.127.59 |
| 10100 (20) | 63.241.55.122 |
| 11000 (24) | 195.71.90.10 |
| 11111 (31) | 66.102.11.99 |

# RD - strukturierte Systeme

- Suche in strukturierten Systemen
  - **für put, delete, lookup muss der zuständige Knoten gefunden werden**
  - **festgelegter Suchpfad anhand der DHT**
    **(kein wahlloses Suchen wie bei unstrukturierten Systemen)**
    **dadurch geringer Suchaufwand**



# Strukturierte Systeme

- Chord
  - **lineare Struktur**
- CAN
  - **n dimensionale Struktur**

# Chord

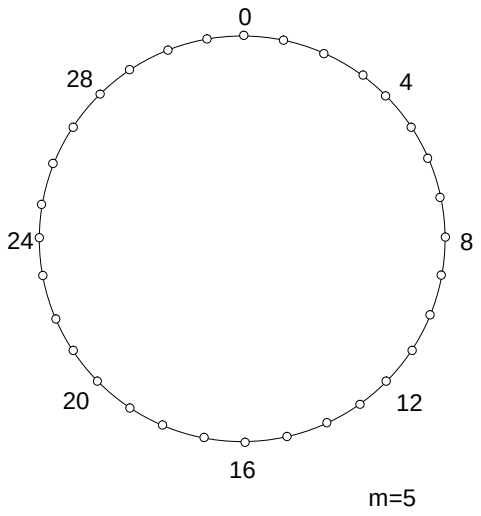
- Speicherung der Daten in DHT
- Hashtabelle ist m-Bit Ringspeicher mit $2^m$ Einträgen (Chord-Ring)
- uniforme Hash-Funktion (möglichst gleich verteilte Schlüssel)
- Objektbezeichner und IP-Adressen der Knoten werden in den selben m-Bit Raum gehasht



# Chord

Zuständigkeit der Knoten für Teilbereiche der DHT:

- hashen der Knoten IP auf m-Bit Wert
  h(ip) → key
  h(62.80.127.59) → 9
- virtuelles Einordnen der Knoten in den Chord-Ring
- jeder Knoten ist zuständig für die Werte, die zwischen ihm und seinem Vorgänger liegen

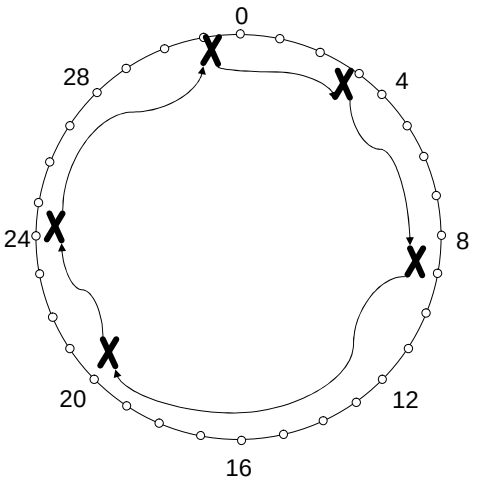


# Chord

Bsp.:
5 Knoten im Netz

| Knoten | IP |
|---|---|
| 00011 (3) | 131.246.120.30 |
| 01001 (9) | 62.80.127.59 |
| 10100 (20) | 63.241.55.122 |
| 11000 (24) | 195.71.90.10 |
| 11111 (31) | 66.102.11.99 |

jeder Knoten kennt
seinen Nachfolger



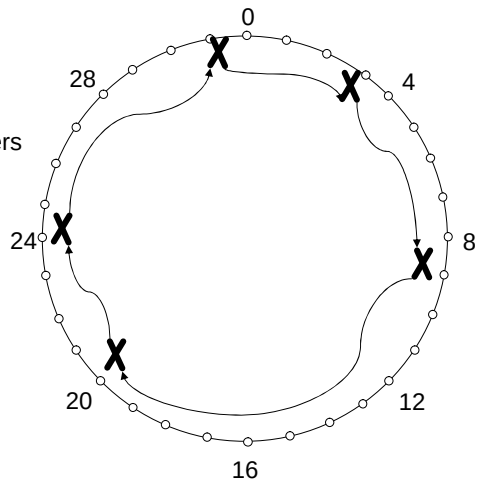

m=5



# Chord

Einfügen von Objekten in
den Chord Ring:

- Objekte haben die Form
  (Bezeichner, Inhalt, IP)
- hashen des Objektbezeichners
  auf m-bit key
  h(Bezeichner) → key
  h(„readme.txt“) → 5
- speichern der IP-Adresse
  auf dem Knoten, der für „5“
  zuständig ist
- Knoten = succ(key)
  62.80.127.59 = succ(5)



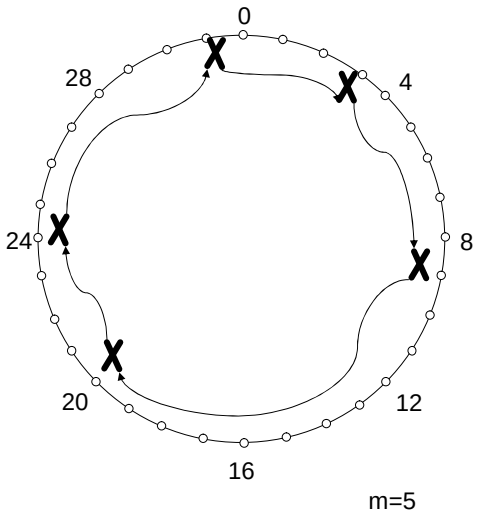

m=5

# Chord

Suche nach einem bestimmten key:

- prüfen, ob der key im eigenen Raum liegt
- senden der Anfrage an den Nachfolger

- langer Suchpfad O(N) bei N Knoten
  → *Finger-Tabelle*






# Chord

- Finger-Tabelle
  - **m weitere Knoten (IPs) werden gespeichert**
  - **i-ter Eintrag hat die Form $succ(n+2^{i-1}) \bmod 2^m$**
    (n ist die Position des eigenen Knotens)
  - **Bsp. für Finger-Tabelle des Knoten 00011 (3)**

| Nr | $succ(n+2^{i-1})$ % m | Knoten |
|---|---|---|
| 1 | $3+2^0=4$ | 62.80.127.59 |
| 2 | $3+2^1=5$ | 62.80.127.59 |
| 3 | $3+2^2=7$ | 62.80.127.59 |
| 4 | $3+2^3=11$ | 63.241.55.122 |
| 5 | $3+2^4=19$ | 63.241.55.122 |

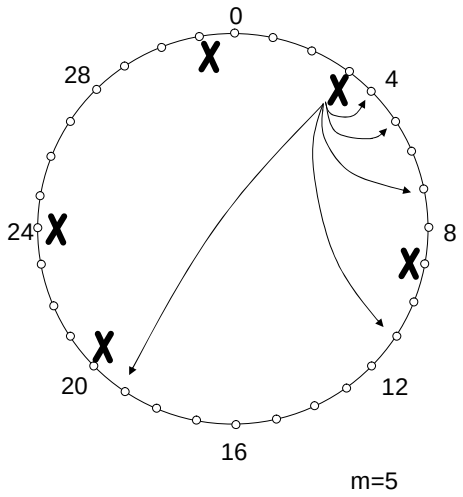
Abstände nehmen exponentiell zu

# Chord

Suche nach einem bestimmten key:

- prüfen, ob der key im eigenen Raum liegt
- prüfen, welcher Eintrag in der Finger-Tabelle von unten annähernd am nächsten liegt
- senden der Anfrage an diesen Knoten

- Kosten Suchpfad O(log N) bei N Knoten






# Chord

- Daten die ein Knoten speichert
  - **Bsp. für Knoten 00011 (3)**

| Hash-Tabelle | |
|---|---|
| 00000 (0) | 62.80.127.59 |
| 00001 (1) | 62.80.127.59 |
| 00010 (2) | 62.80.127.59 |
| 00011 (3) | 195.71.90.10 |

| Finger-Tabelle | |
|---|---|
| 00100 (4) | 62.80.127.59 |
| 00101 (5) | 62.80.127.59 |
| 00111 (7) | 62.80.127.59 |
| 01011 (11) | 63.241.55.122 |
| 10011 (19) | 63.241.55.122 |

# Chord

- Hinzukommen eines neuen Knotens in das Chord-Netz
  - **neuer Knoten n benötig Kenntnis von einem bereits im Netz vorhandenen Knotens n '**
  - **für n wir mit $h(IP_n)$ die virtuelle Position in der Hashtabelle errechnet**
  - **n' muss für n die Finger-Tabelle erstellen**
  - **n muss in die Finger-Tabellen anderer Knoten eingetragen werden**
  - **(key, value) Paare müssen ausgetauscht werden**



# Chord

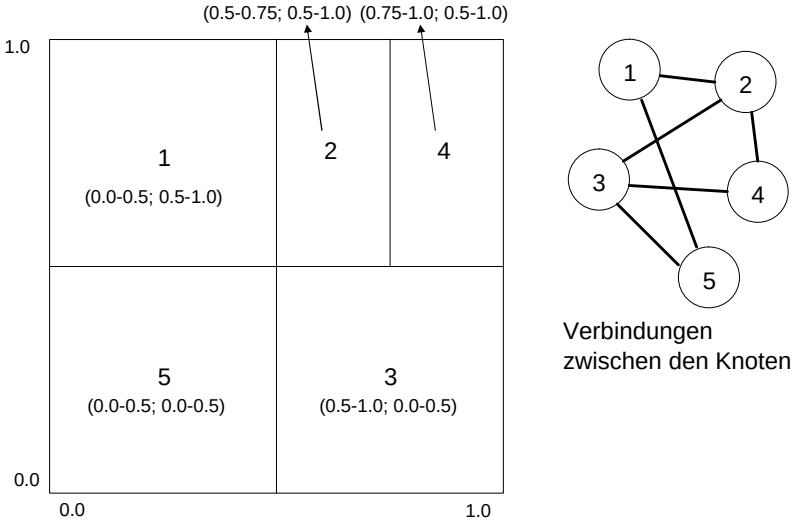
- Suchkosten:
  - **suchen eines Knotens über einen hash-key O(log N)**
    **→ lookup, put, delete**
  - **neuer Knoten im Netz**
    - für m Knoten muss geprüft werden, ob der neue Knoten in deren Finger-Tabellen kommt
      (m ~ log N)
    - log N Knoten müssen gesucht werden
      → $O(\log^2 N)$
    - Annäherung auf O(log N) möglich

# Content Addressable Network

- Speicherung der Daten in DHT
- Hashtabelle ist d-dimensionaler Koordinatenraum (hier 2 dimensional)
- Jeder Knoten ist für eine disjunkte Zone innerhalb des Koordinatenraumes verantwortlich
- Eine Hash-Funktion ordnet jedem Objektebezeichner einem Punkt (x, y) in diesem Raum zu
- Jeder Knoten kennt alle Knoten benachbarter Zonen



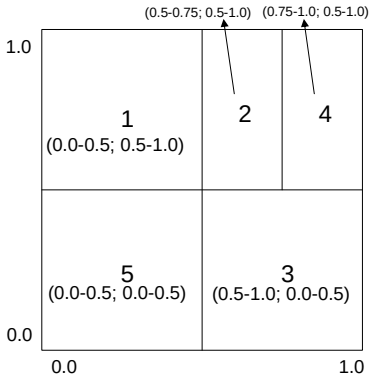
# CAN



# CAN

Einfügen von Objekten in den Koordinatenraum:

- Objekte haben die Form (Bezeichner, Inhalt, IP)
- hashen des Objektbezeichners auf einen Punkt
  h(Bezeichner) → (x,y)
  h(„readme.txt“) → (0,76; 0,8)
- speichern des (key, value) Paares auf dem Knoten, der für (0,76; 0,8) zuständig ist



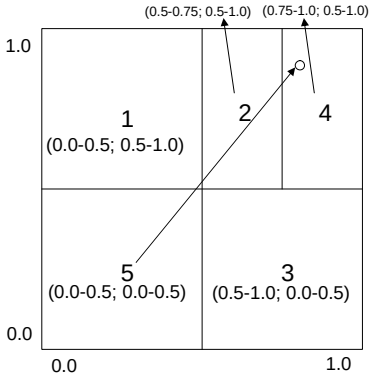



# CAN

Suche nach einem bestimmten Punkt:

- prüfen, ob der Punkt im eigenen Raum liegt
- berechnen, welche Zone den Pfad zum Zielpunkt minimiert
- senden der Anfrage an den für diese Zone zuständigen Knoten

- Suchkosten $O(dN^{1/d})$



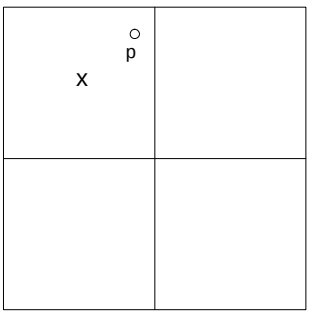
# CAN

- Hinzukommen eines neuen Knotens n in den CAN-Raum
  - **Knotens n ' benötigt**
  - **n ' errechnet zufälligen Punkt p**
  - **berechnen der Nachbarn von n, diese zu den Verbindungen von n hinzunehmen**
  - **die Verbindungen der Nachbarn aktualisieren**
  - **Übertragen der (key, value)-Paare an n**

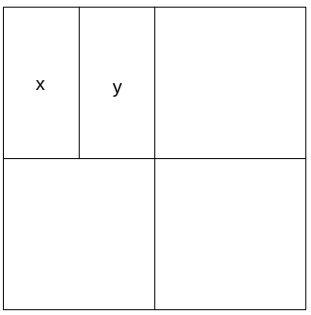


# CAN

- Hinzukommen eines neuen Knotens n in den CAN-Raum



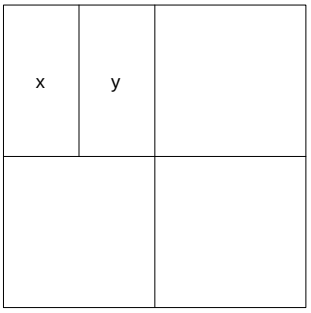

1)



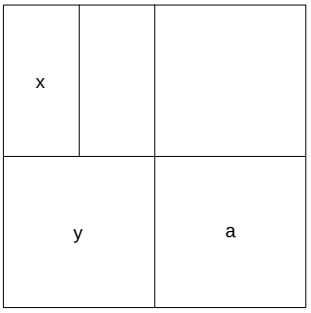

2)

# CAN

- Abmelden eines Knotens
  - **anderer Knoten muss die Zone übernehmen**




1)




2)

  - **übertragen der Tupel an diesen Knoten**

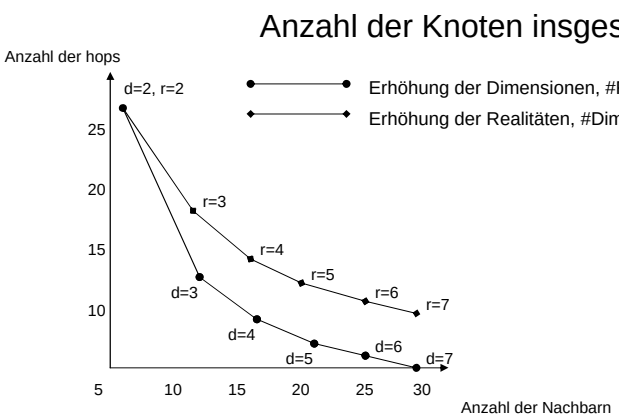


# CAN

- Verbesserungen
  - **Mehrdimensionale Räume Suchkosten $O(dN^{1/d})$**
  - **Mehrere Realitäten**
    - mehrere unabhängige Koordinatenräume
    - jeder Knoten in jedem Raum andere Zone
    - bei Suchanfrage kann jeder Knoten wählen, in welcher Realität der Suchpfad der kürzeste ist

# CAN

- Vergleich Mehrdimensional und mehrere Realitäten






# Komplexe Anfragen

- Komplexe Anfragen mit DHT (Chord)
- Bereichsanfragen mit Näherungslösung (Chord)
- Bereichsanfragen (CAN)
- Mercury (hub)

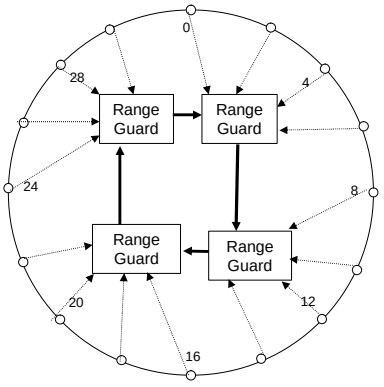
# Komplexe Anfragen mit DHT

- Basiert auf Chord
- Tupel werden direkt in Chord-Ring eingefügt, nicht die IP den Knotens
- m-Attribute -> m Kopien
- Hash-Funktion Reihenfolge-erhaltend
- Bereichsanfragen beschränken sich auf einen Teil des Chord-Rings



# Komplexe Anfragen

- Range Guards
  - **Anfrage in O(k)**



# Bereichsanfragen mit Näherungslösung

- Anfrageergebnisse werden im Chord-Ring gespeichert (Cache)
- Bereichsanfragen [low, high] auf einen key abgebildet
- Hash-Funktion ist lokalitäts-erhaltend
- Neue Anfragen werden aus Cache beantwortet
  (evtl. nur Näherungslösung)

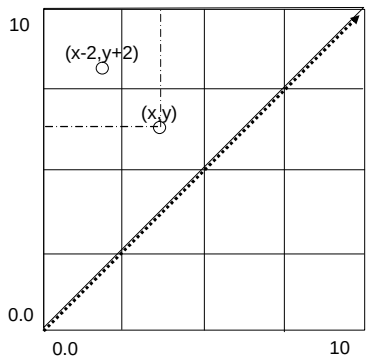


# Bereichsanfragen (CAN)

- 2-dimensionaler Koordinatenraum
- Bereichsanfragen [low,high] auf Punkt (x,y) abgebildet
- Anfrageergebnisse werde gespeichert (Cache)
- Neue Anfragen werden aus Cache beantwortet

# Bereichsanfragen (CAN)

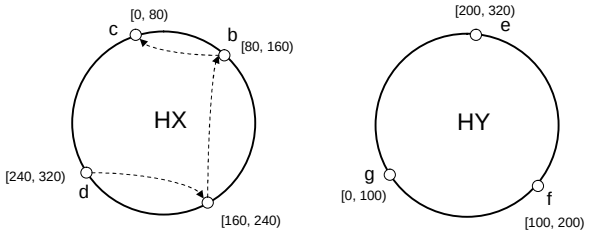
low = x
high = y
➔ x £ y






# Mercury

- basiert auf hubs
- für jedes Attribut existiert ein hub




- Tupel werden direkt eingefügt, nicht die IP des Knotens (m-Kopien)

# Komplexe Anfragen - Vergleich

| | Komplexe Anfragen | Näherungs-Lösung | Bereichs-anfragen | Mercury |
|---|---|---|---|---|
| basiert auf | Chord | Chord | CAN | hubs |
| unterstütze Funktionen | SQL | Bereichs-anfragen | Bereichs-anfragen | Bereichs-anfragen |
| Anzahl Suchschritte | O(N)<br>O(k) | O(log N) | $O(2N^{1/2})$<br>$O(\sqrt{N})$ | O(log N) |
| Kopien pro Tupel | m | 1<br>Cache | 1<br>Cache | m |



# Zusammenfassung

- unstrukturiert / strukturiert
- strukturiert: CAN / Chord
- Verarbeitung komplexer Anfragen