Structured and Unstructured Overlays for Distributed Data

Arvind Krishnamurthy
Yale University

Joint work: Chi Zhang, Anthony Young, Randy Wang

Evolution of Overlays in P2P Systems

Ad-hoc Overlays (Freenet, Gnutella)

Implicit routing
Probabilistic load-balance
Small routing state
Highly scalable systems

Structured Overlays (Chord, Pastry, CAN)

Explicit routing protocol
Medium-scale systems

Unstructured Overlays (Narada, NICE)

Complex Queries
Distribute with locality

Performance-sensitivity
Dynamic changes in quality

?
Motivating Factors

1) Complex data
   - Massive, distributed data collections (web documents, multimedia files)
   - Dynamic data (sensor data, network monitoring, etc.)
   - High-dimensional (or at least multi-dimensional)

2) Complex queries
   - Similarity searches, nearest-neighbor queries
   - Range queries

3) Locality-preserving mapping for better performance

Approach #1: Use DHTs

- Store attributes of a multi-dimensional object in a DHT
- Separate entry made for each attribute to allow for queries on individual attributes
- Can support only exact searches
- Related objects are hashed to arbitrary locations
- Exemplified by PIER; PIER supports arbitrary SQL commands
Approach #2: Hashless DHTs

- Exemplified by Mercury
- Maintain a multi-dimensional index as a set of one-dimensional indexes
- Each index stored in a Chord-like ring without hashing

Data skews require:
1) processor redistribution
2) adjustments to “fingers”

Support for Multi-dimensional Objects

- Maintain a collection of rings for one-dimensional indexes
  - Objects are published in all rings
  - Query sent to only one ring

- Can support complex queries, but incurs storage overhead; not suitable for high-dimensional data
**Approach #3: Hashless CAN**

- Content Addressable Networks (CAN)
  - Manage objects in a multi-dimensional cartesian space
  - Typically, object's position is obtained by hashing its individual attributes

- Store high-dimensional objects in unhashed CAN
  - Related objects are adjacent in the CAN space
  - Each processor assigned a zone
  - Exemplified by pSearch

**Pitfalls of Hashless CAN**

- Problem #1: Dimensionality mismatch
  - Dimensionality of CAN is set to the dimensionality of object space
  - Engineering considerations for CAN:
    - Dimensionality should be typically \( \log(N) \) (where \( N \) is number of processors); typically 10-20
    - Each CAN node has \( 2^d \) neighbors in average
    - CAN node also needs to keep track of neighbor's neighbors for fault-recovery
  - High dimensionality datasets (such as 50 or more) would be impractical to implement using CAN
Pitfalls of Hashless CAN (contd.)

Problem #2: Unbalanced load due to skewed data
- pSearch solution: balance load by having nodes join high-density regions after sampling

- Sampling when a node joins is not sufficient; need to constantly monitor changes in load and repartition space

Pitfalls of Hashless CAN (contd.)

Problem #3: Imbalances in routing state/query processing
- Some zones have a large number of neighbors
- Become popular intermediate nodes for routing and processing queries
Summary of System Requirements

- System should be able to store complex objects
- Be capable of supporting similarity searches, range queries
- Should minimize:
  - Storage overhead
  - Routing state
  - Routing costs
- Should balance:
  - Data distribution
  - Routing state
  - Query/routing costs

SkipIndex: Overview

- Use some hierarchical tree data structure which is used in centralized settings to manage high-dimensional data
- Distribute tree while retaining locality
  - Nearby tree-nodes maintained by nearby processors
  - Processors manage sub-trees
- Do not navigate the tree in a top-down manner
  - Instead jump from one part of the tree to another part of the tree using “long” pointers
- Employ a load-balancing algorithm to ensure that data distribution is balanced
K-D Tree

- Geometrical partitioning of high-dimensional space based on current data distribution

Tree Expansion

- Overloaded regions get split; flexibility in choosing the splitting dimension and position
- Tree could be unbalanced in depth
**Distributing a K-D Tree**

- Take the nodes (or regions) of a K-D tree and distribute it across processors
  - Distributed tree will **not** be navigated top-down
- Consider the nodes of the tree arranged by pre-order traversal

![Diagram of a K-D Tree distribution](image)

**Skip-List**

- Store the nodes of the K-D Tree in a distributed Skip List
- Skip List is a randomized balanced-depth data structure
- Elements are “ordered” at each level of the Skip List
- Can find any item by starting at the top-level skip list and navigating down; at each element a “comparison” is performed

![Diagram of a Skip-List](image)
**Ordering of Regions**

- Consider a region being split into two smaller regions.
- Let R1 appear before R2 in one of the dimensions of the coordinate space.
- Then any subregion of R1 is ordered to be “before” any subregion of R2.
- Two regions can be ordered if their history of region-splits are available.

**Skip Graphs**

- Generalization of a Skip List
  - Adds redundancy for fault-tolerance and to avoid hot-spots.
**Partial Tree**

- Each skip graph element:
  - corresponds to a tree node or region
  - is maintained by a distinct processor
  - keeps track of its split-history
  - maintains pointers to peer elements
- A processor can construct a “partial tree” based on information regarding its peers
- **Routing:**
  - Forward request to a node that is closest to the target region without overshooting it
  - Each element needs to keep track of only log-n neighbors and can perform routing in log-n steps

---

**Distributing K-D Tree Using Skip Graphs**

- Each region is aware only of its peer regions in the skip graph
Routing a Query

- A region routes a query to its peer that is closest to the target query without overshooting the query.

R1 R2 R4

Routing a Query

- R2 is ordered to be “before” the query, while R4 appears “after” the query point in the ordering.
- R1 decides to route the query to R2 and not to R4.
Routing a Query

- R2 peers with R3 which contains the query point
- Has information to route the query to its final destination (R3)

Range Query

- A target range might be refined into multiple smaller ranges and sent to appropriate peers
- Equivalent to performing a “multicast” over the Skip Graph
Load Balance

- Discussion so far assumed that:
  - A region in K-D tree is the same as a Skip Graph node
  - Single regions are assigned to a processor
  - Makes the system sensitive to thresholds used for splitting K-D tree nodes
  - Instead, what we want is to assign a sequence of related regions to a single processor and represent the sequence with a single element in the Skip Graph

Load Balancing Ordered 1-D Keys

- After linearizing the regions of the K-D tree, we are left with a simpler problem
- Dividing an ordered 1D key-space amongst processors in a P2P system is well-studied:
  - Load-stealing approach by Karger-Ruhl
  - Sampling approach used in Mercury
  - Our previous work on balancing Skip Graphs
Components of the load-balancing algorithm

- Processors gossip about current load
  - Gossiping algorithm converges in $O(\log n)$ steps

- A small amount of virtualization is utilized
  - Processors maintain a small number of virtual processors
  - Goal is to keep some number of virtual processors close to system average and the rest are kept idle

- When a virtual processor discovers it is overloaded:
  - It obtains an idle virtual processor and divides its workload with it

- When a virtual processor discovers it is lightly loaded:
  - It offloads its workload to adjacent processors and becomes idle