Advanced Topics in Computer Systems, CS262B
Prof Eric A. Brewer

Querying the Internet with PIER

April 13, 2004

I. Background

Equijoin: join in which the two sides have equal keys

Semijoin: use one relation to prune not-matches from another (like a filter or like selecting the subset of tuples that has a match in the other relation)

Bloom filter:

- start with a bit map of 2^n bits, all zero
- given an object, X, produce k * 2^n hash bits (e.g. use k hash functions with 2^n bits)
- This gives you k indices into the bit map
  - On a read, if all bits are 1, the we have a "hit", else "miss"
  - On a write, simply mark the k bits = 1 (some may already be 1)
- Can't delete an object
- May have false positives
  - With good hash functions, storing C objects. Let z = (C*k)/(2^n), then density d= 1 - e^(-z)
  - Prob(false positive) = d^k
  - If you know target C, pick n and k such that d = 1/2
  - To limit false positives to f, k = ceil(-log2(f)) and n = ceil(log2(Ck))

II. Principles

Relaxed consistency

organic scaling

data in situ ("natural habitat")

standard schemas -- use the schemas implied by widely deployed software

III. DHT-based Joins

Symmetric hash join (SHJ):

- build temp table on each relation, and hash into the other side (symmetrically)
- need to rehash the tables on the join key

- R and S are spread about the DHT; each node scans for local matchs (with selection and projection)
- matches are sent to Q, which is a temp table, marked with whether the result is from R or S
  - all tuples for a given join key thus go to one node, which will build the two tmp tables locally (one for R and one for S)
  - on arrival store into one table and hash into the other to find matches

Fetch Matches:

- assume S is already hashed on the join key
- scan R and issue get for each tuple into S
- apply projection/selection after getting matches then forward

Symmetric Semijoin:

- idea: don't want to rehash both tables (lots of bandwidth to move everything around)
- project both S and R locally to [join key, resourceID]
- use SHJ to compute a Q that has only the join keys that match, which is hopefully smaller than R and S (but need not be)
- from Q issues Fetch Matches to actually get the tuples from R and S (using resourceIDs)

Bloom joins:

- compute a local bloom filter for each of R and S
- create a new tmp namespace for each, BR and BS (assume j nodes per table)
- send local tables to BR and BS, where they are OR'd together
  - we now have a bloom filter on all of R in BR, and likewise for S in BS
- multicast BR to S nodes, and BS to R nodes
- on receipt of the filter, rehash only those tuples that pass the filter
- use SHJ to complete the join
- may have false positives, but they will be culled by the SHJ

IV. Results

Need to have lots of computation nodes to keep bandwidth/node reasonable

All queries start with a multicast, Bloom filter needs two (to distribute BR/BS)

Sym semi-join wins (because it avoids moving a lot of tuples that won't be in the join)

V. Discussion

Aggregation?

Range predicates (vs hashing)

Declarative language?

Quality of results?

Change in the API?

- enable local filtering by passing predicates in the get()
- System R did this to reduce calls into the storage layer