Caching for Client/Server Database Architectures

CPS 296.1
Topics in Database Systems

Roadmap

- Semantic (query) caching

- Object caching (piggyback on queries)
  - Haas et al. “Loading a Cache with Query Results.” VLDB, 1999

Client/Server Database Architectures

- Data shipping
  - Client performs query processing on a local, cached copy of the data retrieved from the server
  - Server primarily services faults
  - Used mostly by object-oriented database systems (OODBMS)
  - Able to exploit client resources

- Query shipping
  - Client sends queries to the server and receives results back
  - Server performs query processing
  - Used mostly by relational database systems (RDBMS)
  - Less able to exploit client resources

Client-side caching

- Traditional approaches to client-side caching for data-shipping architectures
  - Page caching
  - Tuple caching

- New approach that combines data-shipping and query-shipping ideas
  - Semantic caching

Page caching

- Client
  - Caches a collection of index/data pages
  - Processes queries over cached index/data pages
  - Faults in missing index/data pages from the server
  - Manages cache using LRU/LFU/MRU policies
  - Looks just like a DBMS buffer manager

- Server
  - Services page faults and returns pages to client
  - A.k.a. “page server”

Tuple caching

- Client
  - Caches a collection of tuples (objects)
  - Performs indexed scans like page caching
  - Performs non-indexed scans in two alternative ways
    - Ignores cache; sends constraint to server
    - Scans local cache; sends constraint/tuple list to server
  - Manages cache using LRU/LFU/MRU policies

- Server
  - Services tuple faults
  - Performs constrained scans and filters result tuples according to tuple lists provided by client
Semantic caching

- Key idea: in addition to caching query results, remember the queries that generated these results
  - Provides an accurate, semantic description of the content of the cache
  - Supports semantic grouping of the cache content
  - Enables the testing of whether a query can be answering completely by the cache
  - Allows the formulation of a query to retrieve the exact set of missing result tuples from server

Semantic regions

- Cache is managed as a collection of disjoint semantic regions, each including
  - Constraint formula
    - Or really, a materialized view definition (limited to selection view in this paper)
    - Example: \( \sigma \text{salary} < 100K \text{ AND age} < 30 \) Employee
  - Tuple count
  - Collection of cached tuples
  - Replacement data
  - The entire content can be expressed as the disjunction of all constraint formulas

Answering queries

- Say cache content is described by constraint formula \( V \)
- Each single-table selection query is split into two
  - Probe query: (query condition) AND \( V \)
    - Processed over cached data (no need to re-apply \( V \))
  - Remainder query: (query condition) AND (NOT \( V \))
    - If satisfiable, sent to the server and processed there
    - Can be processed in parallel
- Example
  - \( V = ((\text{salary} > 100K \text{ AND age} < 30) \text{ OR age} > 60) \)
  - \( Q = (\text{salary} > 50K) \)

Managing semantic regions

- A query splits an intersecting semantic region into two
  - One is the intersection of query and region
  - The other is the difference with respect to the query
- Results of the remainder query also form a region
  - Number of regions potentially grows exponentially!
- Coalesce policy
  - Always coalesce two regions with same replacement value
  - Never coalesce
  - Heuristic: coalesce two regions with same replacement value only if either is < 1% of the cache size
    - Avoid coalescing big regions, which create big holes when replaced

Semantic cache replacement

- Replace semantic regions with lowest replacement value
- LRU: exploits temporal locality of reference
  - Replacement value = last access time
- Longest Manhattan distance: exploits spatial locality (semantically defined) of reference
  - Replacement value = – (Manhattan distance to the center of the most recent query)

Qualitative comparison

- Data granularity
  - Page caching: coarse; tied to a particular clustering of tuples
  - Tuple caching: fine; high overhead of cache management
  - Semantic caching: semantic grouping; adapts to query load
- Remainder query vs. faulting
  - Page and tuple caching: faulting
  - Semantic caching: remainder query; small messages and parallelism
- Cache replacement policy
  - Page and tuple caching: traditional spatial/temporal locality
  - Semantic caching: + semantic locality
Simulation environment

- Simulation environment
  - Disk modeled in detail
  - Overlap of disk I/O and network transmission considered
  - CPU cost of cache management not modeled (!)
    - Semantic caching may have higher management cost?
- Workload
  - Simple selection queries, whose size varies
  - 90% of the queries are “centered” around a region containing 10% of the data; rest of the queries are distributed uniformly
    - This distribution tends to favor Manhattan distance

Selection on indexed clustered column

- Tuple/LRU: high overhead means fewer tuples fit in cache
- Page/LRU: good spatial locality (because of clustering)
- Semantic/LRU: cache utilization falls for bigger queries because of bigger semantic regions (and therefore holes)

Selection on indexed, non-clustered column

- Page/LRU, MRU: both suffer from non-clustered data
- Tuple/LRU: can adapt better to non-clustered accesses
- Semantic/LRU: semantic grouping offsets non-clustered data; lower overhead

Selection on both columns

- Tuple/LRU: high overhead
- Page/LRU: low compared with tuple/LRU
- Semantic/LRU: depends on coalescing strategy
  - Never coalesce: too much overhead
  - Heuristic works well

Selection on unindexed, non-clustered column

- Page: not shown; suffers from non-clustered data
- Tuple/LRU: must scan local cache before retrieving missing tuples
- Tuple/ignore cache: goes to server directly
- Semantic/LRU: probe and remainder queries can be processed in parallel

Overhead

Additional experiments

- As expected
  - Manhattan works better than LRU
  - In a mobile navigation application, directional Manhattan works better than LRU, MRU

- Give applications the control of the cache replacement policy
Summary of semantic caching

• Materialized views for caching
• Application-dependent semantic locality

• Big concern: How complicated will the constraint formulas become?
  – Without any simplification, constraint formula for each semantic region grows linear with each query
  – Coalescing reduces the number of regions, but not necessarily the complexity of constraint formulas
  – Semantic caching may be no better than tuple caching
  ➢ DynaMat consciously avoids complex constraint formulas
• And what about joins?

Roadmap

• Semantic (query) caching
  ➢ Object caching (piggyback on queries)
  – Haas et al. “Loading a Cache with Query Results.” *VLDB*, 1999

Background and motivation

• Applications ask queries to select objects and invoke methods on relevant objects
  – Example: find hotels and reserve rooms
    ```sql
    foreach h in (select h.oid from hotels h, cities c
    where h.city_oid = c.oid and c.name = 'Durham')
    h.reserveRoom(1, '2002-04-01', '2002-04-03')
    ```
  ➢ Query results alone are insufficient for method invocation

• In traditional client-server systems
  – Queries are executed by clients and servers
  – Methods are executed by clients with caching
  ➢ Query processing is independent of caching

Solution

• Load cache as a by-product of queries
• Introduce cache operators in a query plan to copy objects during query execution
  – A cache operator requires its subplan to preserve objects in their entirety (i.e., no projection)
  – The cache operator then removes from its output stream any attributes that are irrelevant to the query
  – Above the cache operator, the plan is free to perform any additional projection
• Extend the query optimizer to add cache operators into a query plan
Tradeoffs

• What to cache?
  – Cost of the cache operator must be smaller than the savings obtained by it
• When to cache?
  – Late in the plan, so only relevant objects are cached
  – Early in the plan, so other operators are not affected
    • Cache operators may increase the cost of lower operators by forcing them to process objects in entirety

Expensive late caching

Early caching: cheap join:

```
(foreach h in (select oid from …)) h.reserveRoom(…)
```

Late caching: expensive join:

```
foreach h in (select oid from …) h.reserveRoom(…)
```

Alternative approaches

• What to cache (determining candidate collections)
  – Perform dataflow analysis of the application program
  – Analyze SELECT clause of the query; cache if oid is returned
• When to cache
  – Heuristic: caching at the top
  – Heuristic: caching at the bottom
  – Cost-based approach

Caching at the top

• Policy
  – Cache all candidate collections
  – Cache no irrelevant objects
• Algorithm
  – Start with the original plan
  – Place a cache operator at the top of the plan
  – Push down the cache operator through non-reductive operators (so only relevant objects are cached)
Caching at the bottom

- **Policy**
  - Cache all candidate collections
  - Increase cost of other operators as little as possible

- **Algorithm**
  - Start with the original plan
  - Place a cache operator on every leaf that accesses a candidate collection
  - Pull up cache operators that sit below pipelining operators (e.g., filters or nested-loop joins, but not sort)
    - Pull-up reduces the number of irrelevant objects that are cached without increasing the cost of pipelining operators

Cost-based cache operator placement

- **Try to find the best possible plan**
  - Introduce cache operators only if they are beneficial
  - Find the best locations for cache operators in the plan
  - Join order and site selection depend on caching

- **Extend a Selinger-style query optimizer**
  - Enumerate all plans with/without caching
  - Estimate cost and benefit of cache operators
  - Extend pruning condition for dynamic programming

Enumerating all caching plans

- In addition to enumerating normal (thin) subplans, enumerate caching (thick) subplans

<table>
<thead>
<tr>
<th>Join at server:</th>
<th>Cache(Hotels, Cities)</th>
<th>Cache(Hotels)</th>
<th>Join</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hotels Select</td>
<td>Join</td>
<td>Hotels Select</td>
<td>Join</td>
</tr>
<tr>
<td>Cities</td>
<td>City</td>
<td>Cities</td>
<td>City</td>
</tr>
<tr>
<td>Cache(Hotels)</td>
<td>Join</td>
<td>Cache(Hotels)</td>
<td>Join</td>
</tr>
<tr>
<td>Hotels Select</td>
<td>Join</td>
<td>Cities</td>
<td>City</td>
</tr>
<tr>
<td>Cities</td>
<td>City</td>
<td>City</td>
<td>City</td>
</tr>
<tr>
<td>Hotels Select</td>
<td>Join</td>
<td>Cities</td>
<td>City</td>
</tr>
<tr>
<td>Cities</td>
<td>City</td>
<td>City</td>
<td>City</td>
</tr>
</tbody>
</table>

Costing cache operators

- **Overhead of a cache operator**
  - Cost to probe hash table for every object
  - Cost to copy objects that are not yet cached
  - Cost to perform projection

- **Benefit of a cache operator**
  - Relevant objects are not re-fetched
  - Savings depend on
    - Cost to fault in an object
    - Number of objects in the query result that are actually fetched by the application
  - Cost = overhead – benefit
  - Only operators with cost < 0 are useful

Summary of approaches

- **Heuristics**
  - Simple to implement
  - Very little additional optimization overhead
  - Poor plans in certain cases

- **Cost-based**
  - Very good plans
  - Huge search space slows down optimization

Single-table query

- Plain caching still helps a lot
- Heuristic and cost-based approaches return the same caching plan
Three-table join

<table>
<thead>
<tr>
<th>No Caching</th>
<th>Q1(large)</th>
<th>Q1(medi)</th>
<th>Q2</th>
<th>Q3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Traditional caching</td>
<td>405.5</td>
<td>405.5</td>
<td>842.7 129.9</td>
<td></td>
</tr>
<tr>
<td>Caching at the top</td>
<td>71.5</td>
<td>71.4</td>
<td>49.8 177.5</td>
<td></td>
</tr>
<tr>
<td>Caching at the bottom</td>
<td>70.6</td>
<td>425.8</td>
<td>54.2 141.9</td>
<td></td>
</tr>
<tr>
<td>Cost-based caching</td>
<td>71.4</td>
<td>71.4</td>
<td>33.2 130.7</td>
<td></td>
</tr>
</tbody>
</table>

- Cost-based approach consistently picks better plans
  - But notice that for each individual case, cost-based caching never produces the best plan!
- For Q3, caching is just not beneficial

Query optimization time

- Heuristics: very little overhead
- Cost-based: very high overhead—could be higher than the cost of faulting in objects
  - “Meta-optimization” is needed

Summary

- Piggyback on queries to load an object cache
- Not clear whether cost-based approach is better than simple heuristics even when the number of tables in the join is small
  - One strategy is to pick the best of the three plans
    - No caching
    - Caching at the top
    - Caching at the bottom
  - This strategy would consistently beat cost-based approach according to the experiments presented in the paper