Moving Away From Query-at-a-Time Processing

SharedDB and Cooperative Scans

Albert Wu and Sam Fishman
CS 265
The Problem

- Pre-existing query engines have been optimized and evaluated around the **Query-at-a-Time** model.
- In many instances, queries can share both work and data, but don’t.
- Example: `SELECT * FROM users WHERE name = ?`
- More pathological example: $Q_1$ and $Q_2$ both need to scan `users`, but $Q_1$ is much faster than $Q_2$
- Software License Agreements (SLAs): want to guarantee some latency bound for $N\%$ of queries
Why Is This Problem Important?

Web workloads

Machine learning jobs
Existing Solutions – Scanning Procedures
Existing Solutions – Operator Sharing

- Usually designed for OLAP workloads
- Common subexpression (CSE) detection – expensive!
- CJoin/DataPath: dedicated algorithms (joins, sorts, etc.) to enable sharing without CSE

QPipe: Simultaneous pipelining
Sharing Misses

A query Q begins execution at time $T_S$ and completes at time $T_C$.

_Data Sharing Miss._ At time $T_r$, Q requests page $P$, which was previously referenced at time $T_P$. If the request results in a page fault, and $T_S < T_P$, the page fault is a data sharing miss.

_Work Sharing Miss._ At time $T_W$, Q initiates new computation by running operator $W$. If was $W$ also executed between $T_S$ and $T_W$, then there is a work sharing miss.

Two Potential Solutions

Cooperative Scans: How it works

Instead of enforcing sequential scans in round-robin style, always choose to read the chunk that is most “relevant”
Cooperative Scans: How it works

- CScan - range or set of ranges that need to be accessed by a query
- Active Buffer Manager - maintains set of CScan operators

```python
main()
  while (true)
    query = chooseQueryToProcess()
    if query == NULL
      blockForNextQuery()
    continue
    chunk = chooseChunkToLoad(query)
    slot = findFreeSlot(query)
    loadChunk(chunk, slot)
    foreach q in queries
      if (chunkInteresting(q, chunk) and queryBlocked(q))
        signalQuery(q, chunk)
```
Cooperative Scans: How it works

chooseQueryToProcess() chooses the query with maximum “relevance”

```
queryRelevance(q)
    if not queryStarved(q)
        return -∞
    return - chunksNeeded(q) +
    waitingTime(q) / runningQueries()
```

chooseChunkToLoad() chooses the chunk with maximum “relevance”

```
loadRelevance(c)
    return numberInterestedStarvedQueries(c) * Q_{max}
    + numberInterestedQueries(c)
```
Cooperative Scans: Row-Store Experiments

- Used TPC-H, an ad-hoc dataset that mimics business decision support
- Varying buffer size
- Mixes of queries that are more and less CPU-intensive
- Different selectivities of queries
- Scaling data volume
- Concurrency of queries
Cooperative Scans: Row-Store Experiments

Contrasts in access pattern
Cooperative Scans: Row-Store Experiments
Cooperative Scans: Row-Store Experiments

- CPU-intensive query set:
  - Number of I/O requests
  - System time (sec)
  - Average normalized query latency

- I/O-intensive query set:
  - Number of I/O requests
  - System time (sec)
  - Average normalized query latency
Cooperative Scans: Row-Store Experiments
Cooperative Scans: Column-Stores

- More complex implementation because column data can have different sizes
- Uses more of the buffer than row-store implementation
- Reduced data reuse between queries
- Results: still works well!
Cooperative Scans: Issues and Next Steps

- Dynamically-sized chunks
- Further work on column-stores
- Integration into real database systems
- Working on cooperative scans for main memory
  - See: Qiao et al. Main-Memory Scan Sharing For Multi-Core CPUs, PVLDB ‘08.
SharedDB: Enabling Operator Sharing

**Key idea:** augment each intermediate tuple with a query_id set

(Simplified) Q2:

SELECT * FROM users, orders WHERE u.id = o.user_id AND order.price > ?
SharedDB: Generating a Query Plan

- Queueing happens at each node in the graph (besides queries)
- Request all queries up front (PreparedStatements)
- How do we come up with this global plan?
  - A: Merge individual plans
SharedDB: Shared Joins

**Merge:**
1. Add predicate on `query_id`
2. (Nothing else)
SharedDB: Shared Joins

<table>
<thead>
<tr>
<th></th>
<th>query_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>1,2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>query_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

We can filter out the first row by adding the query_id constraint.
SharedDB: Extending to All Operators

- Step 1: Compile each query separately
- Step 2: For each query, for each operator:
  - If there is already an operator instance related to the appropriate tables or intermediates, union into selection
  - Otherwise, add the new operator to the plan

For some operators (e.g. Top-N and Group-By), there must be a separate query-specific step
SharedDB: Experiments
SharedDB: Experiments
SharedDB: Next Steps

- Global query optimizer
  - See: Giannikis, et al. Shared Workload Optimization, PVLDB 2014

- Automatic NUMA-awareness

- Distributing query plan among machines

- Issues:
  - Latency bound vs. individual query performance
  - Need all queries beforehand
Wrap-up

- Data sharing vs. work sharing
- Cooperative Scanning and SharedDB are orthogonal ways of improvement, rather than competing ones
- Issue: overheads from scheduling