Multi-Query Optimization

S. Sudarshan
Computer Science and Engineering Dept
I.I.T. Bombay
sudarsha@cse.iitb.ernet.in
http://www.cse.iitb.ernet.in/~sudarsha

Joint work with: Prasan Roy, S. Seshadri, Siddhesh Bhobhe

Overview

- Database Query Languages and Query Optimization
- Issues in common subexpressions
- DAG representation of alternative query evaluation plans
- Simple heuristics
- Greedy heuristic and optimizations
- Performance study

Relational Databases

- Data represented in the form of tables
- Employee data: relation *emp*

emp-name	*dept-name*	*age*	*sal*
Sudarshan	CSE	33	100,000
Phatak	SIT	50	400,000
Prasan	CSE	25	108,000
Arvind	SOM	24	180,000

- Department data: relation *dept*

dept-name	*dept-addr*	*head*
CSE	CSE Bldg, IIT B	Dhamdhere
SIT	Math Bldg, IIT B	Phatak
SOM	SOM Bldg, IIT B	Korgaonkar

Database Query Languages

- Database query languages are declarative
- Philosophy: say what you want, not how to get it
- Example: SQL

```
Q1:  select emp.name, dept.addr
     from   emp, dept
     where  emp.dept-name = dept.dept-name

Q2:  select dept-name, avg(salary)
     from   emp
     where  age < 35
     group by dept-name
```

Query Optimization

- Relational algebra: an algebra on relations

  select: $(\sigma_{age<35}(emp))$

  join: $(emp \bowtie_{emp.dept\text{-}name=dept.dept\text{-}name} dept)$

  project: $(\Pi_{name}(emp))$

  aggregation: $({}_{dept\text{-}name}\mathcal{G}_{sum(sal)}(emp))$

- User level languages translated to relational algebra

- Multiple expressions for same result (similar to $(a+b)*c = a*b + a*c$)

- E.g. $(r \bowtie s) \bowtie t \equiv r \bowtie (s \bowtie t)$

- Different expressions have different costs

- Query optimizer responsible for finding best expression

Why Multi-Query Optimization

- To tackle complex decision support queries
  - Sequence (batch) of related queries
  - Queries with common subexpressions
    - Esp. with use of views and WITH clause
    - Multi-output operators
      E.g. Engage.Fusion
- Data warehouses, OLAP, ...

Multi-Query Optimization

- Given: query, or a batch of queries
- Goal: exploit common subexpressions
- Problem: many alternative query plans with different costs
- E.g.:
  - $Q_1 : (R \bowtie S) \bowtie P$ and $Q_2 : (R \bowtie T) \bowtie S$ – no common subexpressions
  - Rewrite Q_2: $(R \bowtie S) \bowtie T$ – now $\exists$ common subexpression
  - Can compute $R \bowtie S$, store (materialize) and reuse
- Problem: expression form with CSE may or may not be cheaper!
- Problem: lots of alternative plans $\Rightarrow$ expensive

Steps in MQO

- Recognize possibilities of sharing
  - Extend Volcano's Query-DAG generation algorithm
- Find best plan taking sharing into account
  - Local best plans may not be globally optimal
  - Exhaustive algorithms are doubly exponential
  - Need good heuristics

Related Work

- Plain Query Optimization
  - System R: bottom-up
  - Volcano: top-down, extensible
- Early MQO Papers

  Sellis et al. [TODS88], Park and Segev [ICDE88], Cosar et al. [CIKM93]
  - Not integrated with basic optimizer tightly enough, inefficient
- Common Subexpression Elimination

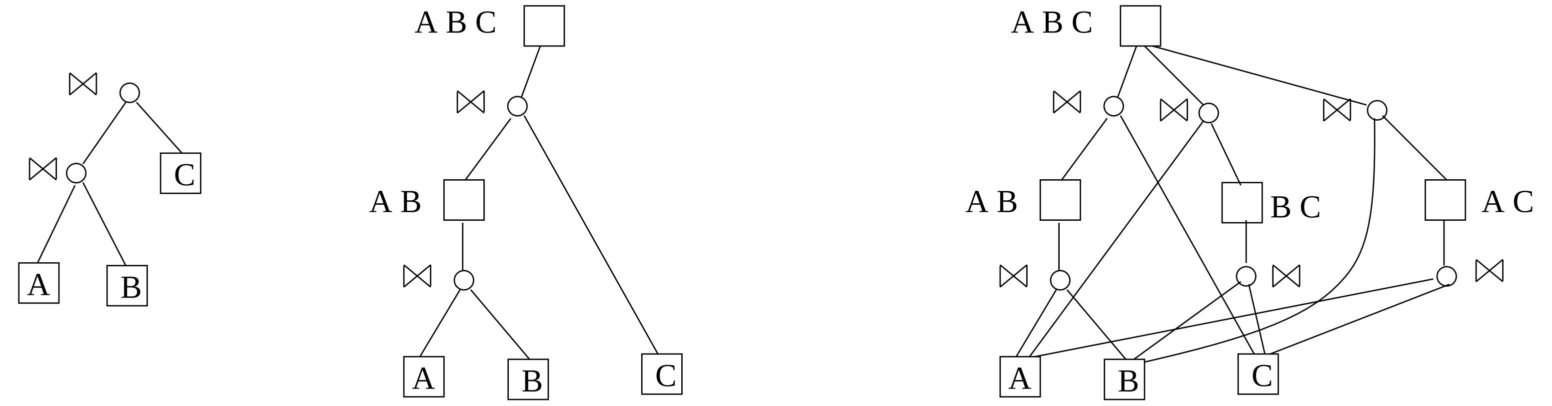
  Shivakumar and Subramaniam [SIGMOD98]
- Invariants in Nested Queries

  Rao and Ross [SIGMOD98]

Related Work (contd.)

- Materialized View/Index Selection
  Roussopoulus [TODS82], Ross et al. [SIGMOD96], Gupta [ICDT97], Labio et al. [ICDE97], Yang, Karlapalem and Li [VLDB97]
  - Needs to handle updates
  - MQO is a subproblem
- Optimization in Presence of Materialized Views
  Chaudhari et al. [ICDE95]

AND-OR DAG Representation



(Commutativity not shown - every join node has another join node with inputs exchanged, below the same equivalence node)

(a) Initial Query (b) DAG representation of query (c) Expanded DAG after transformations

- □: *equivalence* nodes
- ○: *operation* nodes

DAG Generation

- Apply transformations
  - E.g: $(E_1 \bowtie E_2) \bowtie E_3 \equiv E_1 \bowtie (E_2 \bowtie E_3)$
    $\sigma_{E_1.A=5}\ (E_1 \bowtie E_2) \equiv \sigma_{E_1.A=5}(E_1) \bowtie E_2$
  - Result of a transformation on E is equivalent to E.
    Added under the same equivalence node
  - New equivalence nodes may be created
  - Hashing based scheme for detecting duplicate derivations used in Volcano (a.k.a hash consing)
  - We use above for detecting common subexpressions
- Physical DAG models physical properties e.g. sort order

Detecting Repeated/Common Subexpressions

- Hashing scheme of Volcano
  - gives ids to equiv. nodes
  - hash function on (op, ids-of-inputs) help locate duplicate operations
  - detects identical expressions / (duplicate derivations)
  - transformations create new equiv nodes iff expression created is new
  - Equiv. node parents of identical expressions different $\Rightarrow$ common subexpression
- Additional *unification* step merges equivalence classes

Expression Subsumption

- Given: $\sigma_{A<5}(E)$ and $\sigma_{A<10}(E)$
  - $\sigma_{A<5}(E) \quad \equiv \quad \sigma_{A<5}(\sigma_{A<10}(E))$
- Given ${}_{dno}\mathcal{G}_{sum(sal)}(E)$ and ${}_{age}\mathcal{G}_{sum(sal)}(E)$
  - ${}_{dno}\mathcal{G}_{sum(sal)}(E) \quad \equiv \quad {}_{dno}\mathcal{G}_{sum(sal)}({}_{dno,age}\mathcal{G}_{sum(sal)}(E))$
  - ${}_{age}\mathcal{G}_{sum(sal)}(E) \quad \equiv \quad {}_{age}\mathcal{G}_{sum(sal)}({}_{dno,age}\mathcal{G}_{sum(sal)}(E))$
- Bottom-up traversal of the DAG adds new subsumption derivations (and new equivalence nodes)

Volcano Search Algorithm

- Finds local best plan recursively, ignoring sharing possibilities
- Uses costlimit for a directed search (branch & bound)
- Cost of operation node = local cost + sum of input costs
- Cost of equivalence node = **min** cost of children
- Best plan of equivalence node:
  lowest cost child, plus plans for its inputs

Volcano-SH

- Run plain Volcano on queries to find local best plans
- Find common subexpressions amongst local best plans
- Choose to materialize and share a common subexpression if:

$$(\text{mat-cost} + \text{num-uses} \times \text{reuse-cost}) < (\text{num-uses} \times \text{recomp-cost})$$

- Note: recomp-cost changes depending on materialization of descendants
- May choose to materialize and share only part of a common subexpression
- Number of uses depends on which shared parents are materialized

Volcano-RU

- Given $Q_1, \ldots, Q_n$
- When optimizing Q_i
  - Find best local plan, with special case for equivalence nodes that are in best local plans for $Q_j, j < i$:
    * set cost of node to **min** ($\text{recomp-cost}, \text{mat-cost} + \text{reuse-cost}$)
- Note: Once a node is chosen to be materialised, mat-cost is not added for further sharing.
- After optimizing all Q_is, run Volcano-SH to find and share further common subexpressions (within queries)

The Greedy Heuristic

1. Compute DAG and best plans for each node using Volcano
2. Find *benefit* of materializing each unmaterialized node
   (a) mark a node n as materialized
   (b) set cost of node n = reuse-cost of n
   (c) recompute best plans for all nodes with above cost
   - this step done *incrementally*

   (d) total-cost = recomp-cost of n + mat-cost of n +
   sum best costs for root of DAG
   (e) benefit of n = reduction in total cost due to materializing n
   (f) unmark n, and unset cost of node n
3. Choose node with max cost benefit,
   mark it, materialize it and set its cost to reuse-cost
4. Repeat from Step 2, until no decrease in cost

Incremental Cost Update Propagation

- Recomputing best plans from scratch on each benefit computation very expensive
- Observation: *Marking a node as materialized only affects its ancestors*
- Incremental Propagation:
    1. Start from marked node, and propagate cost changes up the DAG
    2. If cost of a node does not change, no need to propagate changes further up
- Use topological sort numbering to decide which node to propagate changes to next (*ensures no revisits*)

Improvements to Greedy: Monotonicity Heuristic

- **Monotonicity Assumption:** Benefit of materializing a node never increases as other nodes are materialized.
- Monotonicity $\Rightarrow$ *old benefit estimates are overestimates on current benefit*
- "old" $\equiv$ computed in an earlier iteration of greedy
- *Key idea*: Keep heap of nodes sorted on benefit *overestimates*
  - Repeat forever
    1. Pop top node t
    2. If benefit t is ≤ 0, break
    3. if benefit of t is *current*, t is the node with maximum benefit; delete from heap and materialize it
    4. else update its benefit, mark as *current*, reinsert in heap
- Monotonicity generally holds; reduces opt. run time by 90%

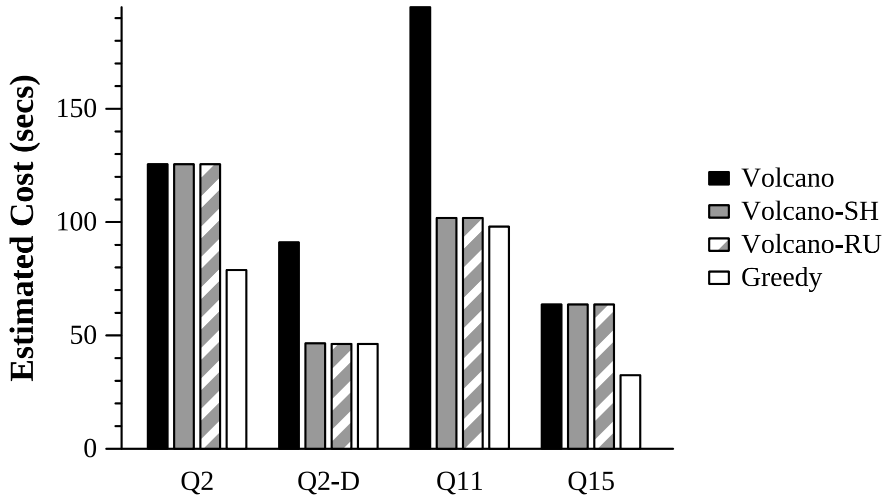
Extensions

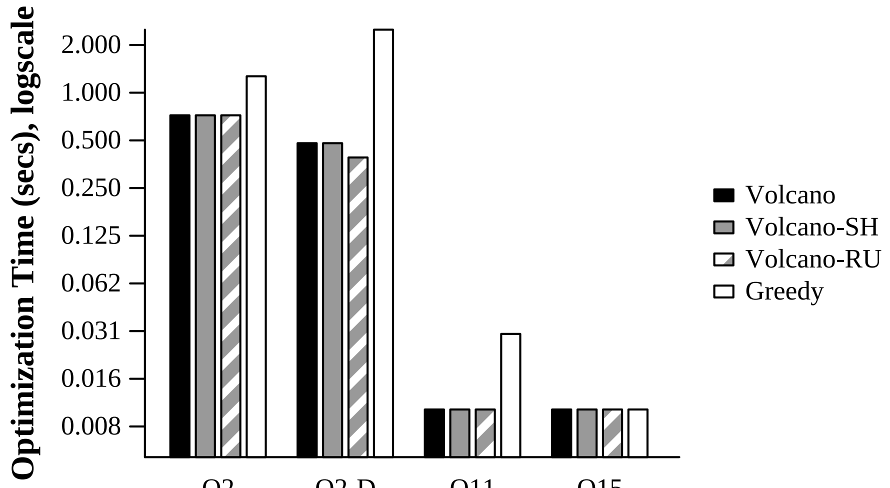
- Index Selection
- Nested Queries
- Parameterized Queries with common subparts
  e.g. multiple calls on query/stored procedure with different parameter values
- Space Constraints
  Greedy with benefit per unit space

Performance Study

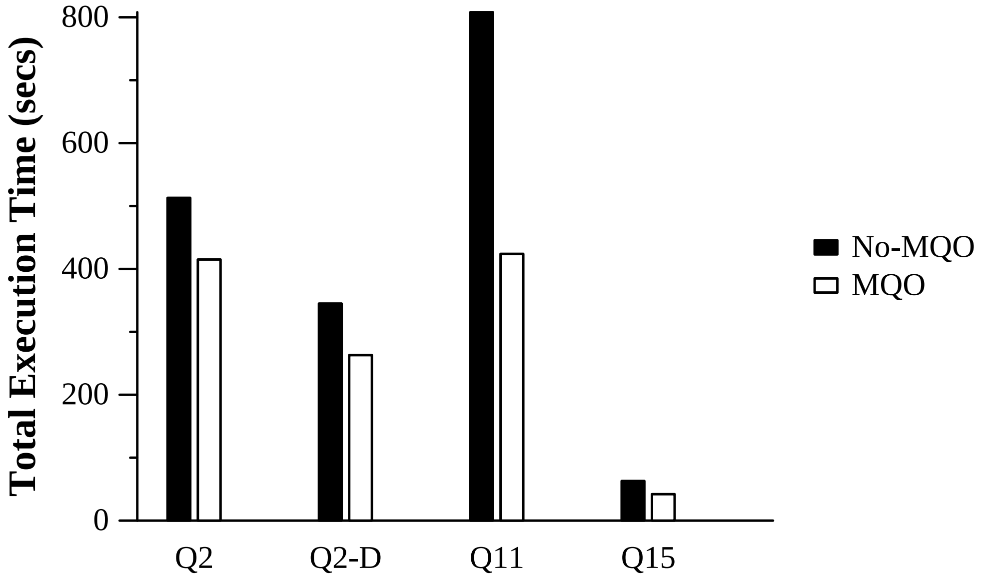
- Implemented all algorithms (17K lines Volcano + 3K lines MQO)
- Benchmarks
  - Stand-Alone TPCD
  - Batched TPCD
  - Scaleup

Performance of Stand-Alone TPCD

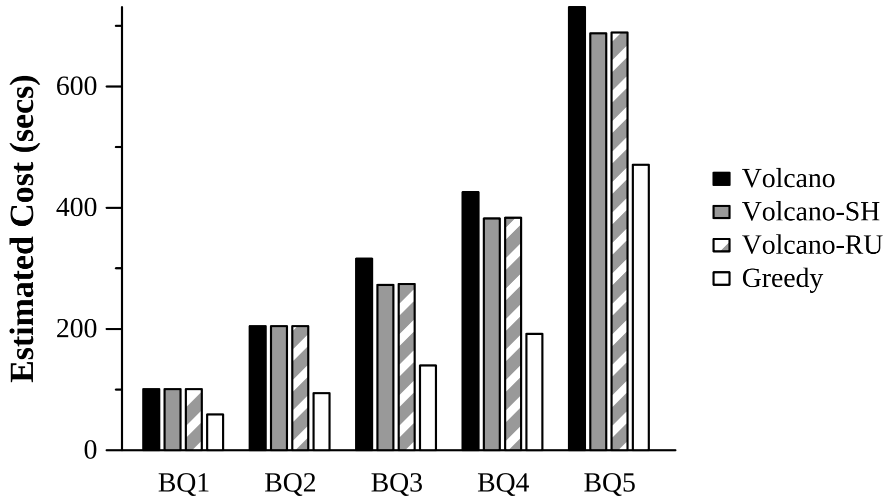


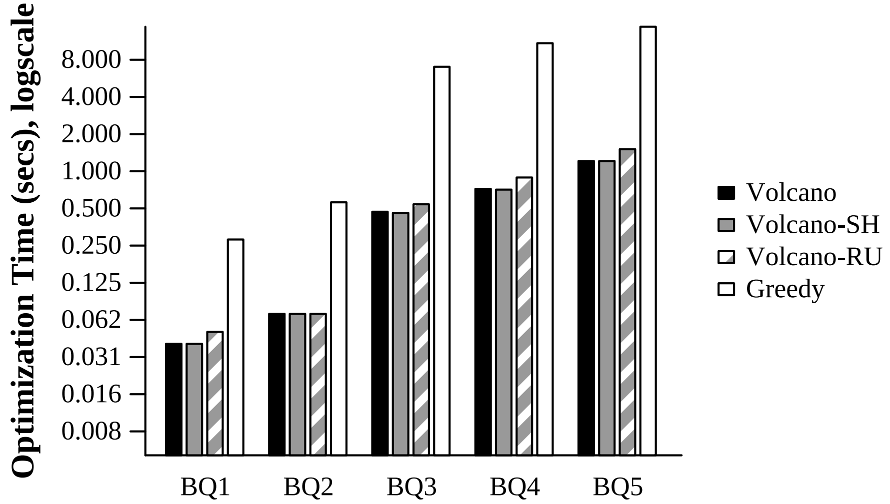


Performance on SQL-Server 6.5

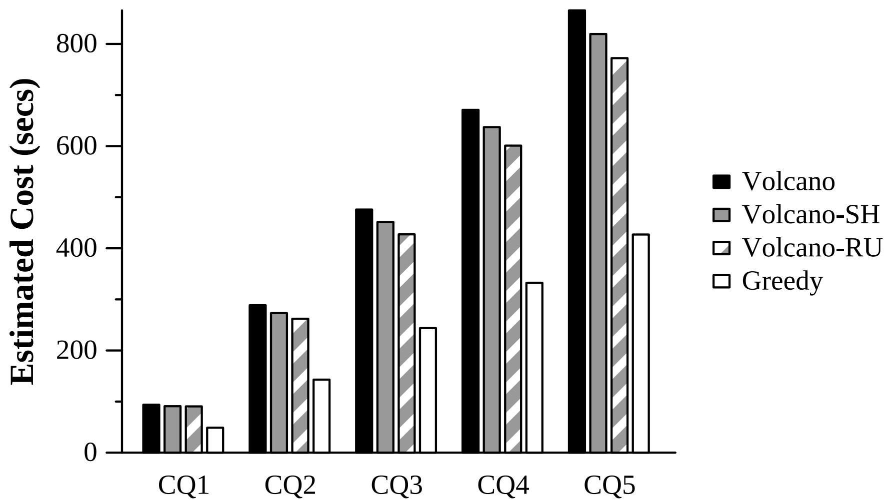


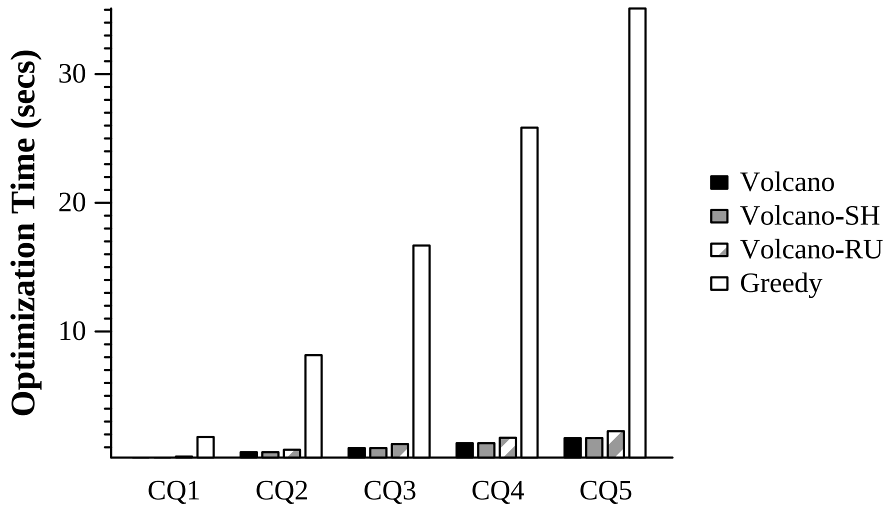
Performance of Batched TPCD



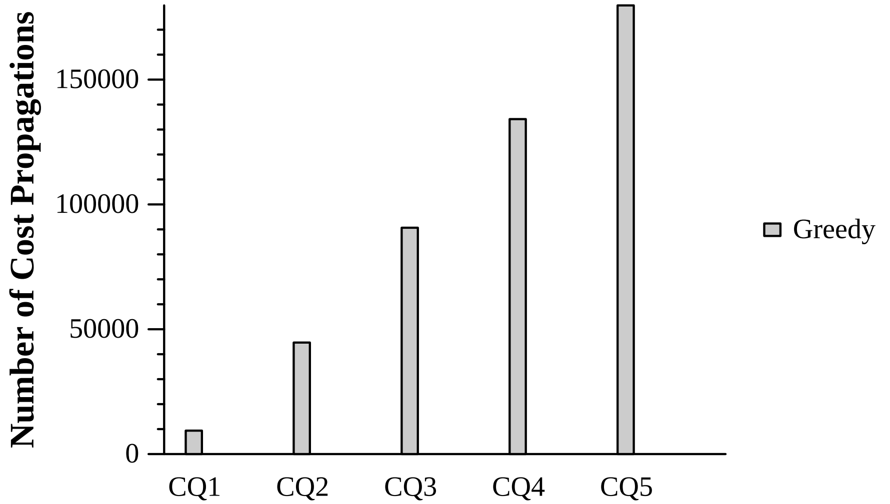


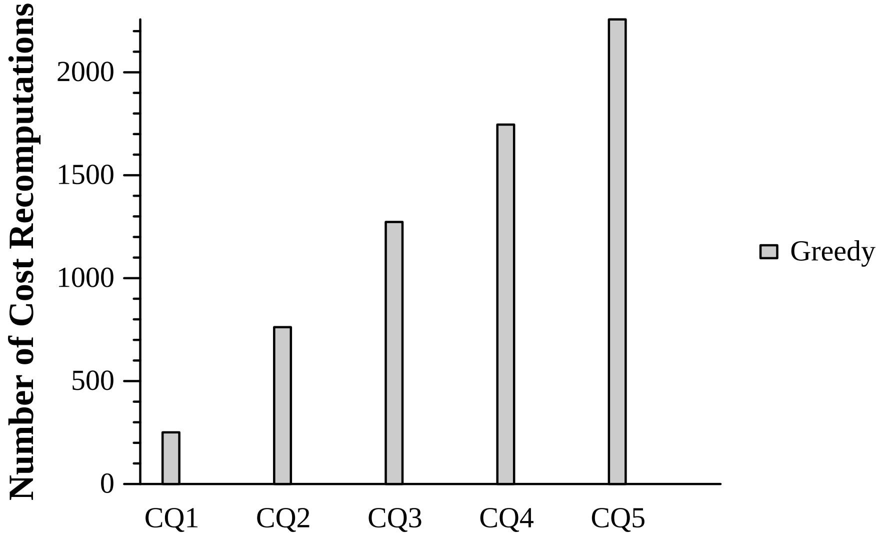
Scaleup Analysis





Empirical Complexity of Greedy





Conclusion

- Greedy is best overall, followed by Volcano-RU and Volcano-SH
- Volcano-SH, Volcano-RU have neglegible overheads.
  Overhead for Greedy acceptable for complex expensive queries
- Greedy shows linear scaleup with number of queries
- Benefits show up on TPCD benchmark running on Microsoft SQL-Server 6.5
- Advise:
  Use Volcano-RU if query very cheap and syntactically complex.
  Use Greedy for all other cases

Future Work

- Space Constraints
- Query Caching
  - Sequence of queries
  - Need to predict future
  - Decide what to maintain in the cache
- Large workloads: Query Abstraction
- Scheduling – try to avoid materialization
- View/Index Selection: Update Costs