An Overview of Query Optimization in Relational Systems

Surajit Chaudhuri
Microsoft Research
surajitc@microsoft.com
http://research.microsoft.com/~surajitc

What to expect from this tutorial?

- Query Optimization in practice
  - Framework
  - A few key ideas
  - Active areas of work
- No cool theorems
- Provide a perspective that helps place your work in a systems context

Why Query Optimization?

- SQL is a high level language ("declarative")
  - Physical data independence
- Needs to be compiled into a program over relational query engine
- Query optimization compiles the query into a program that takes the "least" resources
  - Acid test of data independence
Outline

- Preliminaries
  - Relational query engine
  - "Programs" over relational query engines (operator trees)
- Query Optimization Framework
- System R optimizer
- Modern Optimizers
- How to interact with Optimizers
- Active Areas of work
- Conclusion

Relational DBMS Components

Storage Structures

- Tables
- Indexes
  - Columns
    - Single column, Multiple columns
  - Type
    - B+ indexes, Bitmap indexes, Hash indexes
  - Clustering
    - Clustered, Non-clustered
  - Implied "index-evaluable" predicate
Implementation Operators for Scan and Selection

- **Scan([index], table, predicate)**
  - Sequential Scan
  - Indexscan: Which index(es) to use?
  - Always push down “index-evaluable” predicates
- **Filter(table, predicate)**

Implementation Operators for Join

- **Join([method], outer, inner, join-predicate)**
  - Asymmetric
  - Effect of physical properties of input streams (e.g., sorted input)
  - Physical properties of output stream (e.g., sorted)
  - Pipelined v.s. Blocking
    (Nested Loop v.s. Sort-Merge)

Join Operators

- **Join(Sort-Merge, R1, R2, R1.a = R2.a)**
  - Can exploit sorted order on R1.a
  - Output is a sorted order
  - Blocking
- **Join(Nested-Loop, R1, R2, R1.a = R2.b)**
  - Sorted inputs of no consequence
  - Output has the same sort order as R1.a
  - Pipelined
Generic View of Operators

- Input: One or more data streams
- Output: One data stream
- Implementation
  - open()
  - getnext()
  - close()
- Pipelined/Blocking

Operator Trees

- An algebraic expression tree consisting of selection and join can be realized
  - using an operator tree consisting of scan, filter and join nodes
  - root node is the output of algebraic expression
  - leaf nodes are scans on stored relations
  - child node is an input data stream to its parent
- (Sequential) Operator tree same as
  - annotated Query Tree
  - execution Plan (or, simply plan)

Example of an Operator Tree
Execution of an Operator Tree

- Demand-driven architecture is the simplest
- open() is propagated from the root
- getnext() at the root is propagated
- If getnext() at the root fails to return a new tuple, then no more answers for the query

Properties of Trees

- Edge properties
  - Size of the data stream
  - Physical properties (e.g., sorted order)
- Node properties
  - Cost of an operator
  - Pipelined v.s. blocking
- Cost of tree = sum of costs of nodes
- How to estimate the edge and node properties?

Outline

- Preliminaries
- Query Optimization Framework
- System R optimizer
- Modern Optimizers
- How to interact with Optimizers
- Active Areas of work
- Conclusion
Goal of Query Optimization

- Multiple ways to compile a SQL query over the relational engine
  - Algebraic properties
  - Implementations for each operator
  - Costs of the alternatives may be widely different
- Find the program with least cost
  - Query optimization as a planning problem?

A Framework for Query Optimization

- Equivalence Transformations
  - Algebraic properties
  - Implementation options
- Estimation Model
  - Needs to estimate cost of an operator tree (incrementally)
- Search Algorithm
  - Fast, Memory-efficient

Outline

- Preliminaries
- Query Optimization Framework
- System R optimizer
- Modern Optimizers
- How to interact with Optimizers
- Active Areas of work
- Conclusion
SPJ Queries

Select A.a, B.b, C.c
From A, B, C
Where A.x = B.x and B.y = C.y
Order By A.a

Algebraic Transformations

- Select and Join commute
  - Filter(Join(A,B), a) = Join(Filter(A,a), B)

- Joins are associative and commutative:
  - Join(Join(A,B), C) = Join(Join(B,A), C)
  - Join(Join(A,C), B) = Join(Join(A,B), C)
  - Many equivalent expressions

- Linear join trees (restricted use of AC properties)

Implementation

Transformations

- Scan
  - B+ tree index scan
  - (Sargable) Predicate: Between and its degenerate forms

- Filter
  - Any Boolean expression

- Join
  - Sort-Merge, Nested-loop, Indexed Nested-loop
**Estimation Model**

- **Goal:** Estimate the cost of an operator tree
  - Number of tuples, Number of distinct values, cost of sub-expressions
- **System-R used a bottom-up computation.**
  - For every node:
    - Computes these parameters of the operator for the given parameters of the input data streams
    - Derives properties of the output data streams
- **Propagates estimates up the tree**
  - For base tables, this information is computed by "run statistics"

**Deriving Statistics**

- Consider a "normal" form of SPJ query:
  \[ Q = \text{Filter} (\text{Cartesian-Product}(R_1, \ldots, R_n), f) \]
- Selectivity is fraction of data that satisfies predicate
  - \[ \text{Size of } Q = \text{Selectivity}(f) \times \text{Size-of}(R_1) \times \ldots \times \text{Size-of}(R_n) \]
- **Compute selectivity of a filter expression**
  - (a) Determine selectivity of atomic predicates using statistics \((a > 3, a=b)\)
  - (b) Derive the selectivity of a Boolean expression from (a)

**Selectivity Estimates for Atomic Predicates**

- **Selections**
  - Column = v
    - \( F = 1/(\text{#column}) \)
  - Column Between [a1,a2]
    - \( F = (a2-a1)/(\text{Hkey} - \text{Lkey}) \)
- **Joins**
  - Column1 = Column2
    - \( F = 1/\max(\text{#column1}, \text{#column2}) \)
Selectivity Estimates for Boolean Expressions

- **P1 AND P2**
  \[ F(P1 \text{ AND } P2) = F(P1) \times F(P2) \]

- **NOT P1**
  \[ F(\text{NOT } P1) = 1 - F(P1) \]

- **P1 OR P2**
  \[ F(P1 \text{ OR } P2) = F(P1) + F(P2) - F(P1) \times F(P2) \]

- **Interesting issue:**
  There are multiple ways to derive statistics for the same expression

---

Cost Estimates

- **What to measure?**
  - Throughput
  - IO cost + \( w \times \) CPU cost
  - IO cost = Page Fetches

- **Examples of Scan cost**
  - \( S \): \# of Pages(R)
  - \( CI \): \( F \times (\# \text{ of Pages}(R) + \# \text{ of Index Pages}) \)
  - \( NCI \): \( F \times (\# \text{ of Tuples}(R) + \# \text{ of Index Pages}) \)

- **Interesting Issue**
  - Effect of database buffers?

---

Cost Estimates (Join)

- **Nested Loop Join**
  - Cost-of(N1) + Size-of(N1) * Scan-cost(N2)
  - Scan-cost(N2) depends on indexes used

- **Sort-Merge Join**
  - Sort(N1) + Sort(N2) + Scan(Temp1) + Scan(Temp2)
Search Strategy

- Need to order joins (linearly)
- Naïve strategy:
  - Generate all n! permutations of joins
- Prohibitively expensive for a large number of joins
  - Overlapping subproblems, use of optimal substructures
  - Ideal for dynamic programming

Dynamic Programming

- Goal: Find the optimal plan for Join(R_1,..R_n, R_{n+1})
  - For each S in \{R_1,..,R_n, R_{n+1}\} do
    - Find Optimal plan for Join(Join(R_1,..,R_n), S)
  - Endfor
  - Pick the plan with the least cost
- Principle of Optimality:
  - Optimal plan for a larger expression is derived from optimal plan of one of its sub-expressions
- Complexity
  - Enumeration cost drops from O(n!) to O(n^2^n)
  - May need to store O(2^n) partial plans
  - Significantly more efficient than the naïve scheme

Example

```
1 2 3 4
1 2 3 12 34
1 2 3 1 2 4 23 41
1 2 3 1 2 3 4
```

© Surajit Chaudhuri  PODS-06-01/05
Search Control Features

- **Avoid Cartesian product**
  - Defer all Cartesian products as late as possible to avoid “blow-up”
  - Don’t consider $(R_1 \times R_2) \Join R_3$
    if $(R_1 \Join R_3) \Join R_2$ is feasible

- **Recognize “interesting orders” as violation of principle of optimality:**
  - $\text{Cost-of}(SM(R_1,R_2)) > \text{Cost-of}(NL(R_1,R_2))$
  - But, $\text{Cost-of}(SM(SM(R_1,R_2)), R_3)$ may be much less expensive than other options

Handling Interesting Orders

- Identify all columns that may exploit sorted order (by examining join predicates)
- Collapse into equivalent groups
- One optimal partial plan for each interesting order
- Example:

Key Ideas from System R

- **Cost model based on**
  - access methods
  - size and cardinality of relations
- **Enumeration exploits**
  - dynamic programming
  - one optimal plan for each equivalent expression
  - violation of principle of optimality handled using interesting order
Limitations of System R

- **Cost Model**
  - one aggregate number for every column (inaccurate)
  - independence assumption
- **Transformation**
  - limited to join ordering
- **Enumeration**
  - limited to single block queries

Outline

- Preliminaries
- Query Optimization Framework
- System R optimizer
- Modern Optimizers
  - Cost Estimation
  - Transformations
  - Enumeration Architectures
- How to interact with Optimizers
- Active Areas of work
- Conclusion

Selectivity Estimation Models

- Estimate selectivity by executing the query on a “sampled” database
- Pre-compute Statistical Descriptors
  - Histograms : Range Predicates
  - Frequent Values, Number of distinct values : Equality Predicates

Selectivity Estimation Models

Selectivity Estimation Models

Selectivity Estimation Models
**Histograms for Derived Columns**

- **Filter**
  - Filter acts as a mask
  - Interpolate count in a partial bucket using uniformity assumption
  - Filter with host variables hard to handle
- **Join**
  - “Normalize” two histograms
  - “Join” two histograms
- **Shortcomings:**
  - Cannot capture correlation
  - Month = Jan and Item = Jacket
  - Needs multi-dimensional histograms
  - Not effective for equality queries

**Various Histogram Structures**

- **Equi-depth:**
  - All buckets have same number of values
  - Adjacent values co-located in buckets
- **V-Optimal**
  - Groups contiguous sets of frequencies
  - Minimizes variance of the frequency approximation
  - “Optimal” for a subset of range queries
- **A General Framework [PHJS96]**
  - Assign a metric to each value
  - How to partition the metric space?
  - What information is kept for each bucket?
  - What assumptions are made of values within a bucket

**Building Statistics**

- **Advantage**
  - Optimization sensitive to available statistics
- **Disadvantage**
  - Expensive to collect and maintain
  - “Auto-maintain” statistical descriptors
- **Use of sampling**
  - Must take into account data layout
  - Needs “block” sampling
  - Not effective for number of distinct value
  - How sensitive is optimization to accuracy of statistics?
Outline

- Preliminaries
- Query Optimization Framework
- System R optimizer
- Modern Optimizers
  - Cost Estimation
  - Transformations
  - Enumeration Architectures
- How to interact with Optimizers
- Active Areas of work
- Conclusion

Transformations

- SQL is the target
- SQL identity may not be a good way to think about transformations
  - Use algebraic framework
- May add, not just commute operators
- Finding transformations is easy, finding a good one is hard
  - Broadly applicable
  - Interaction with other transformations

Case Studies of Transformations

- Commuting group by and join
- Commuting join and outer-join
- Optimize multi-block queries
  - Collapse multi-block query to a single block query
  - Optimize across multiple query blocks
Commuting Group By and Join

- Traditionally, execution of group-by follows execution of joins
- "Pushing down" group by past a join:
  - Group By "collapses" an equivalence class
  - Therefore, may reduce cost of subsequent joins
  - Can be pipelined with index scans
- Application needs to be cost based since
  - The cost of group by itself may be increased
  - Access methods on base tables may no longer be useful for the join
- Related to Optimization of Select Distinct queries

Commuting Group By and Join

- Schema:
  - Product(pid, unitprice, ..)
  - Sales(tid, date, store, pid, units)
- Example:

Introducing Group By

- Schema:
  - Sales(tid, date, store, pid,amount)
  - Category(pid,cid)
- Example:
Applicability of Group By/Join Transformations

- Schema constraints, arbitrary aggregation functions
- No schema constraints, but properties of aggregate functions
  - $\text{Agg}(S1 \cup S2) = f(\text{Agg}(S1), \text{Agg}(S2))$
  - May sometime require use of derived columns
- Related to collapsing multi-block queries into a single block query

Multi-Block Queries

- Single Block Query
  - **Select** columns
  - **From** base-tables
  - **Where** conditions
  - **Group By** columns
  - **Order By** columns

- Multi-block structure arises due to
  - views with aggregates
  - table expressions
  - nested sub-queries

- **Divide and Conquer**
  - leverage single block optimization techniques

Example of A Nested Subquery

Select Emp.Name
From Emp
Where Emp.Dept# IN
 (Select Dept.Dept#
  From Dept
  Where Dept.Loc = "Denver"
  AND Emp.Emp# = Dept.Mgr)
Example of A View

Create View DepAvgSal as
(Select E.did, Avg(E.Sal) as avgsal
From Emp E
Group By E.did )

Select E.eid, E.sal
From Emp E, Dept D, DepAvgSal V
Where E.did = D.did
And E.did = V.did
And E.age < 30 and D.budget > 100k
And E.sal > V.avgsal

Merging Nested Subquery

- Think of “IN” as a semi-join between Emp and Dept on
  - Emp.Dept# = Dept.Dept#
  - Emp.Emp# = Dept.Mgr
- Convert Semi-join to Join

  Select Emp.Name
  From Emp
  Where Emp.age < 30 And Emp.Dept# IN
  (Select Dept.Dept#
   From Dept
   Where Dept.Loc = "Denver" And Emp.Emp# = Dept.Mgr)

Result of Merging

Query:
Select Emp.Name
From Emp
Where Emp.Dept# IN
(Select Dept.Dept# From Dept
Where Dept.Loc = "Denver" And Emp.Emp# = Dept.Mgr)

Transformed Query:
Select Emp.Name
From Emp, Dept
Where Emp.Dept# = Dept.Dept#
And Emp.Emp# = Dept.Mgr And Dept.Loc = "Denver"
Nested Subqueries (2)

- Presence of aggregates in the nested sub-query requires careful treatment
- Key Observations:
  - For each outer tuple, create the "count" of matching inner tuple and compare to D.parking
  - If outer matches no inner tuple, then the outer produces an output tuple ("count bug")

```sql
Select D.Name
From Dept D
Where D.parking <=
(SELECT count(E.Emp#)
From Emp E
WHERE E.Dept# = D.Dept#)
```

Merging Nested Subqueries (2)

- Results in a left outerjoin between the parent and the child block (preserves tuples of the parent)
- Outerjoin reduces to a join for sum(), average(), max(), min()
- Transformed Query:

```sql
Select D.Name
From Dept D
LOJ Emp E
On (E.Dept# = D.Dept#)
Select count(E.Emp#) Group By D.Dept#
Having D.parking <= count(E.Emp#)
```

Optimization Across Blocks

- Collapsing into a single block query is not always feasible or beneficial
- We can still optimize by sideways information passing across blocks
- Idea similar to semi-join
  - Outer provides inner with a list of potentially required bindings
  - Helps restrict inner's computation
  - "Once only" invocation of inner for each binding
Example of Query with View

Create View DepAvgSal as:
Select E.did, Avg(E.Sal) as avgsal
From Emp E
Group By E.did

Select E eid, E.sal
From Emp E, Dept D, DepAvgSal V
Where E.did = D.did
And E.did = V.did
And E.age < 30 and D.budget > 100k
And E.sal > V.avgsal

Example of SIP

Select E eid, E.sal
From Emp E, Dept D, DepAvgSal V
Where E.did = D.did
And E.did = V.did
And E.age < 30 and D.budget > 100k
And E.sal > V.avgsal

- DepAvgSal needs to be evaluated only for cases where V.did IN
Select E did
From Emp E, Dept D
Where E.did = D.did
And E.age < 30 and D.budget > 100k

Result of SIP

Supporting Views:
1. Create view ED as (Select E eid, E did, E.sal
From Emp E, Dept D
Where E.did = D.did
And E.age < 30 and D.budget > 100k)
2. Create View LAvgSal as (Select E did, Avg(E.Sal) as avgsal
From Emp E, ED
Where E.did = ED.did
Group By E.did)

Transformed Query
Select ED eid, ED.sal
From ED, LAvgSal
Where E did = ED did and ED sal > LAvgSal avgsal
More Comments on Transformations

- **Summary of Multi-Block Transformations**
  - SIP (semi-join) techniques result in use of views
  - Merging views related to commuting Group By and Join
  - Nested Sub-query => Single Block transformations result in J/OJ expressions
- **SQL semantics is tricky**
- **Applicability conditions are complex**
- **Transformations must be cost based**

Outline

- **Preliminaries**
- **Query Optimization Framework**
- **System R optimizer**
- **Modern Optimizers**
  - Cost Estimation
  - Transformations
  - *Enumeration Architectures*
- **How to interact with Optimizers**
- **Active Areas of work**
- **Conclusion**

Enumeration Architectures

- **Stress on extensibility (for optimizer developers)**
- **Key features**
  - Explicit representation of transformations as rules
  - Explicit representation of “properties” of plans
  - sort-order, estimated costs
  - Rule engine
- **Examples: Starburst, Volcano**
- **Framework != Optimizer**
Starburst v.s. Volcano

- **Starburst**
  - Heuristic application of algebraic transformations
  - “Core” cost-based single-block join enumeration
- **Volcano**
  - No distinction among transformations
  - Cost-based
  - More difficult search control problem

Starburst Overview

- **QGM for representation of queries**
- **Rewrite Rule Engine**
  - Condition -> action rules where LHS and RHS are arbitrary C functions on QGM representation
  - Rule classes for search control
  - Conflict resolution schemes
  - Customizable search control for rule classes
- **Plan Optimizer**
  - Handles implementation alternatives
  - LOLEPOP (operator)
  - STAR (implementation alternatives)
  - GLUE (achieving required properties)

Volcano Overview

- **Query as an algebraic tree**
- **Transformation Rules**
  - Logical rules, Implementation rules
- **Optimization Goal**
  - Logical Expression, Physical Properties, Estimated Cost
- **Top-down algorithm**
  - Sub-expressions optimized on demand
  - An equivalence class table is maintained
  - Enumerate possible moves
    - Implement operator (LOLEPOP)
    - Enforce property (GLUE)
    - Apply Transformation Rules
  - Select “move” based on promise
  - Branch and bound
Distributed Systems

- Optimization in Distributed Systems
  - Communication cost vs. local processing time
- Evolution of Distributed Systems
  - Scalability concerns => Parallel systems
  - Distributed information => Replicated sites

Parallel Database Systems

- Objective is to minimize response time
- Forms of parallelism
  - Independent, Pipelined, Partitioned
- Scheduling of operators becomes an important aspect of optimization
- Can scheduling be separated from the rest of the query optimization?

Parallel Database Systems

- Two step approach:
  - Generate a sequential plan
  - Apply a scheduling algorithm to “parallelize” the plan
- The first phase should take into account cost of communication (e.g., repartitioning cost)
  - Influences partitioning attribute
- Scheduling algorithm assigns processors to operators
  - Symmetric schedule: assigns each operator equally to each processor
  - Suboptimal when communication costs are considered
Outline

- Preliminaries
- Query Optimization Framework
- System R optimizer
- Modern Optimizers
- How to interact with Optimizers
- Active Areas of work
- Conclusion

Interacting with Optimizer

- Information on the plan chosen by the optimizer
  - Showplan (MS), Visual Explain (IBM)
  - Load plan information in tables
- Optimizer hints to control the nature of plans
- Optimization Level
  - How exhaustive is the search for the “optimal” plan? (greedy v.s. DP join enumeration)
- Statistics
  - Update Statistics
  - Manual update to statistics (distinct values, frequent values, highest values)

Optimizer Hints

- Give partial control of execution back to the application developer
- Can specify
  - Join ordering, Join methods, Choice of Indexes
- Liability
  - Hard to maintain as software is upgraded or database statistics changes
- Example
  - Select emp-id
  - From Emp (index = 0)
  - Where hire-date > '10/1/94'
Outline

- Preliminaries
- Query Optimization Framework
- System R optimizer
- Modern Optimizers
- How to interact with Optimizers
- Active Areas of work
- Conclusion

Active Areas

- OLAP
- Optimization for ADT
- Content Based Retrieval
- Old-fashioned problems

OLAP

- Spreadsheet paradigm drives the querying model
- Complex ad-hoc queries over large databases
- Stress on use of
  - Indexes
  - Multi-pass SQL
  - Materialized Views
  - Top-k Queries
  - “Helper Constructs”
  - Data Partitioning, Parallelism
Using Indexes

- **Selection**
  - Use single or multi-column indexes
- **Join**
  - Join indexes, Use two clustered indexes
- **Projection**
  - Use as a vertical projection
- **Group By**
  - On-the-fly aggregation
- **Index AND-ing**
  - data scan for fewer pages
  - avoid data scan altogether
- **How to use the right set of indexes?**

Multi-Pass SQL

- **Backends always cannot digest complex SQL**
- **Middleware (“ROLAP”) tool optimizes SQL generation**
  - Creates and maintains materialized views
  - Tuned to backends
  - Defines appropriate temporary relations

Materialized Views

- **View Definitions**
  - Must consider aggregation as part of view definitions
- **Optimization Problem**
  - Choose an equivalent expression over materialized views and tables
  - Appropriate access methods
- **Reminders**
  - Need for a cost-based choice
  - Multiple materialized views may apply
  - Using base table may be better than using cached results!
  - “2-step” algorithms can be significantly worse
Materialized Views over Star Schema

- Order
  - OrderNo
  - OrderDate
- Customer
  - CustomerNo
  - CustomerName
  - CustomerAddress
  - City
- Salesperson
  - SalespersonID
  - SalespersonName
  - City

Fact table
- OrderNo
- CustomerNo
- SalespersonID
- Product
  - ProdNo
  - ProdName
  - ProdDescr
  - Category
    - CategoryNo
    - CategoryDescr
  - UnitPrice
- Date
  - DateKey
  - Date
  - Month
  - Year
- City
  - CityName
  - State
  - Country

Top K Queries

- Find k best restaurants in Seattle by … where …
- If k is small compared to result size then optimal query plan may be different
  - Use nested loop instead of sort-merge
  - Use non-clustered index scan instead of sort
  - Alternative row blocking techniques
- Commercial databases provide constructs

Dominance among Views

- Use a more specific view that and can answer the query
- Dominance is a partial order
- Need cost-based optimization
  - Consider a query on (category, state)
  - The view on (product, state)
  - dominates (product, city)
  - does not dominate (category, city)
  - (product, state) and (category, city) are candidate materialized views to answer the query
Helper Constructs
- Ensuring “Optimality” of plans not feasible
- Provide constructs in language that help optimizer
  - Does not extend expressivity
  - But, may result in significant performance enhancement
- Example: Each subtotal requires a separate aggregate query

<table>
<thead>
<tr>
<th>MODEL</th>
<th>Sum By</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sum</td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CUBE and ROLLUP
- Rollup (order of columns matters)
  - Group By product, store, city Rollup
    - Group by product, store, city; Group by product, store; Group by product
- Cube (order of columns does not matter)
  - Group By product, store, city Cube
    - One aggregation on each subset of {product, store, city}:
      - Group by product, store, city; Group by store, city; Group by city, product
      - Cube = A set of Roll-up operations

Optimization for ADT
- Independent user-defined functions
  - Select * From Stocks Where stocks.fluctuation > .6
  - Associate a per-tuple CPU and IO cost with udf
  - New issues in enumeration
    - Udfs are harder than selections, but easier than relations
- Relationship among udfs
  - E.g., Spatial datablade supports related spatial indexes
  - Use rules to specify semantic relationships
  - Cost-based semantic Query Optimization
  - New issues in costing and enumeration
    - Don’t generate all equivalent expressions
    - How to use costs uniformly across ADT-s
    - “Mix and match” or “ADT-specific” optimization?
Content Based Retrieval

- Fuzzy matches
  - Associate a degree of match with selection
- Top k fuzzy matches
  - Only interested in “top 10” matches with a suspect’s sketch
  - Match may involve multiple features
  - How to exploit the specification of for reducing the cost of data access?
  - Related to near neighbor search
- Relationship to IR work

Old-fashioned Problems

- Compile Time vs. Run time optimization
  - Choose plan and Exchange
- Resource governor
  - Adapting optimization to memory constraints
- Sensitivity of the cost model
  - How detailed a cost model needs to be?
- Client-Server issues
- Object models

Concluding Remarks

- Many factors determine performance
  - Query Processing engine
  - Query Optimizer
  - Physical database design
  - Settings of the “knobs”
- Many open problems
  - Architectural framework is important
  - Oversimplification may render results useless
  - Need to pay attention to SQL semantics
  
  surajitc@microsoft.com
  http://research.microsoft.com/~surajitc