Extensible Query Processing in Starburst

Michael DiBernardo
September 26th, 2006
Motivation

- It’s 1989
- Databases are hard to use for domain-specific applications
Motivation

• e.g. If I want to build an app for biological data mining
  • I have to write my own predicates for sequence similarity
  • I have to convert VARCHARs to DNA/RNA representations in the application
  • This results in lots of “glue code” and data selection logic spread across db and application layers
Motivation

- Secondary motivation:
  - Authors desire an arena in which they can invent many facetious names for things
Proposed Solution

- Make the system extensible, and allow “database customizers” with domain-specific knowledge to tailor

- Extensibility was a hot topic because of emergence of OO databases
Proposed Solution

• Extensions:
  • Language extensions (new datatypes and operations)
  • Data management extensions (access and storage)
  • Processing extensions (new joins, query transforms)
• To support all these, need a powerful language processor
Anatomy of Starburst

- Design of query language
- Internal representation of a query
- Query rewriting
- Cost-based optimization
Outline

• Design of query language

• Internal representation of a query

• Query rewriting

• Cost-based optimization
Query language

- Derivative of SQL called “Hydrogen”
- Add hooks for language extensions
- Increases orthogonality
  - Emphasize table as fundamental unit
  - Views are tables
  - Expressions evaluate to tables
Query language

• Keeps language simple, but there are many ways to describe the same query
  • Makes optimization difficult
  • Makes understanding queries difficult
• Also implies that queries can be verbose
Query language

• Keeps language simple, but there are many ways to describe the same query
  • Makes optimization difficult
  • Makes understanding queries difficult
• Also implies that queries can be verbose
Outline

• Design of query language
• Internal representation of a query
• Query rewriting
• Cost-based optimization
Goal is to have a flexible representation that is easily extensible, sufficiently expressive, and easy to operate on.
Query Graph Model

- Tables (normal and derived) are boxes with heads
- Table accesses are vertices
- Predicate applications are edges (with rectangles, for some reason)
Example

```sql
SELECT partno, price, order_qty
FROM quotations Q1
WHERE Q1.partno IN
  (SELECT partno
   FROM inventory Q3
   WHERE Q3.onhand_qty < Q1.order_qty
     AND Q3.type = 'cpu')
```
Example
Example

```
SELECT partno, price, order_qty
FROM quotations Q1
WHERE Q1.partno IN
  (SELECT partno
   FROM inventory Q3
   WHERE Q3.onhand_qty < Q1.order_qty
   AND Q3.type = 'cpu')
```
Example
Example

```sql
SELECT partno, price, order_qty
FROM quotations Q1
WHERE Q1.partno IN
    (SELECT partno
     FROM inventory Q3
     WHERE Q3.onhand_qty < Q1.order_qty
     AND Q3.type = 'cpu')
```
Example
Outline

- Design of query language
- Internal representation of a query
  - Query rewriting
- Cost-based optimization
Query Rewriting

• First component of ‘optimization’

• Basic idea:
  
  • Maintain a set of rules that recognize a condition and perform a transformation on QGM

• e.g. If you have two SELECTs (boxes) and they handle dups the same, merge them into one box
Query Rewriting

- Three classes of rules
  - Predicate migration
  - Projection push-down
  - Operation merging
- DBC can add to or create new classes
Query Rewriting

• “Rule engine” applies rules

• Selects rule to apply based on some ordering (sequential, statistical, priority)

• Uses forward chaining to generate rewrites

• Has a “budget” to prevent it from running too long
Query Rewriting

- What if there’s more than one rule that works?
- We don’t have a notion of cost here
- So generate all alternatives and push them forward
- Maintains modularity at expense of performance
Outline

• Design of query language

• Internal representation of a query

• Query rewriting

• Cost-based optimization
Cost-based estimation

- The general framework we’re familiar with:
  - Generate a space of plans from the QGM
  - Estimate the cost of each plan
  - Search the space
  - Each operation is optimized independently
Plan generation

- Build a plan using a grammar-driven approach

- Terminals are “low level plan operators” (LOLEPOPS)
  - i.e. relational algebra primitives implemented as functions that operate on and produce tuples

- Nonterminals are “strategy alternative rules” (STARs)
  - i.e. higher-order functions that combine nonterminals and LOLEPOPS
Plan costing

- Each table (normal or derived) has a set of properties
- Properties are modified on the fly as LOLEPOPS are applied to them
- Three kinds:
  - Relational (e.g. tables joined, columns accessed)
  - Operational (e.g. ordering of tuples)
  - Estimated (e.g. cardinality)
Search Strategy

• Nothing especially exciting here

  • Alternative STARs are ranked for pruning

  • Change queueing to implement different searches

  • Other parameters

    • e.g. Turn off bushy trees
Wrapup

- Starburst =
  - Orthogonal and extensible query language
  - Generic query representation (QGM)
  - Rule-based query rewriting
  - Plan optimization w/ grammar-driven generation
  - Lots of silly subcomponent names
Discussion