ECS 165B: Database System Implementation
Lecture 21

UC Davis
May 14, 2010
Class Agenda

• Last time:
  – Views and Relational Encodings of XML (2)
  – DavisDB Part 4 Sneak Preview

• Today:
  – DavisDB Part 4 Overview and Architectural Cookbook Session

• Reading:
  – Ch 15 of Ramakrishnan and Gehrke
Announcements

• Grades for Part 2 have been sent out

• Reminder: Part 3 due Sunday @ 11:59pm

• Don't forget to turn off any printf debugging statements before submitting (they interfere with shell output)
  – At this point, you should be using a debugger, not printfs, anyway!

• Project Part 4: out tonight
DavisDB, Part 4: Query Evaluation Engine
DavisDB, Part 4: Query Engine

- Culmination of the project: you'll implement a query engine for a fragment of SQL consisting of four commands:
  - Queries: select-from-where
  - Updates: insert into, delete from, update

- As in Part 3, SystemParser handles the front-end, you implement the back-end: a class called QueryEngine

- Relatively low bar for getting full credit
  - Optimization not required
  - Starter code and architectural template provided

- Opportunities for extra credit
  - e.g., Query optimizer
The Select Command: Syntax

```
select <attrName>, ... , <attrName>
from <relName>, ..., <relName>
[ where <attrName> <cmpOp> <attrOrValue> and ... and
  <attrName> <cmpOp> <attrOrValue> ] ;
```

- Each attribute name `<attrName>` must be **fully-qualified**
  - e.g., `select R.A, S.B` rather than `select A,B`
- No self-joins (no duplicates in `<relName>` list)
- `where` clause is optional
  - `<cmpOp>` one of `<`, `>`, `=`, `<>`, `<=`, `>=`
  - `<attrOrValue>` either a fully-qualified attribute name, or a constant:
    - quoted string like "abc", "xyz", ...
    - integer like 2, -1
    - float like 2.0, -1.7  (floats **must** contain a decimal point)
The Select Command: Semantics

- Execute the query, according to standard SQL semantics
- Parser calls QueryEngine::select to execute:

```cpp
ReturnCode select(
    int nAttributes, const RelationAttribute attributes[],
    int nRelations, const char* relations[],
    int nConditions, const Condition conditions[]
);
```

- Must check that query typechecks wrt the database schema
- Print results to console output using SystemPrinter, as in `print` command from Part 3
- How to implement? We'll discuss in a moment...
The Select Command: Example

```
select Sailors.sid, Sailors.sname
from Sailors
where Sailors.rating > 5 and Sailors.age < 30.0;
Sailors.sid int, Sailors.sname char(32)
--------------------------------------------------
32,Andy
71,Zorba
2 records total.
=> RC_OK
```
The Insert Command: Syntax and Semantics

```
insert into <relName> values (<value>, ..., <value>) ;
```

- Values specified as in other commands:
  - quoted string like "abc", "xyz", ...
  - integer like 2, -1
  - float like 2.0, -1.7 (floats must contain a decimal point)

- Semantics: calls QueryEngine::insert to perform the insertion

```
ReturnCode insert(const char* relName, int nValues,
                 const TypedValue values[]);
```

- Must verify that it typechecks wrt the database schema; don't forget to update indices too!
The Delete Command: Syntax

```sql
delete from <relName>
[ where <attrName> <cmpOp> <attrOrValue> and ... and 
<attrName> <cmpOp> <attrOrValue>] ;
```

- Each attribute name `<attrName>` must be *fully-qualified*
- where clause is optional
  - `<cmpOp>` one of `<, >, =, <=, >=`
  - `<attrOrValue>` either a fully-qualified attribute name, or a constant:
    - quoted string like "abc", "xyz", ...
    - integer like 2, -1
    - float like 2.0, -1.7  (floats **must** contain a decimal point)
The Delete Command: Semantics

- Invokes QueryEngine::remove to delete all tuples matching the specified conditions (or all tuples, if none specified)

```
ReturnCode remove(const char* relName,
                  int nConditions,
                  const Condition conditions[]);
```

- Must verify that query typechecks wrt schema; don't forget to update indices!

- Looks a bit like a selection query, with a deletion operation on top? Bear this in mind when designing execution engine...
The Update Command: Syntax

```
update <relName>
set <attrName> = <attrOrValue>
[ where <attrName> <cmpOp> <attrOrValue> and ... and
  <attrName> <cmpOp> <attrOrValue> ] ;
```

- Each attribute name `<attrName>` must be `fully-qualified`
- `where` clause is optional
  - `<cmpOp>` one of `<`, `>`, `=`, `<>`, `<=`, `>=`
  - `<attrOrValue>` either a fully-qualified attribute name, or a constant:
    - quoted string like "abc", "xyz", ...
    - integer like 2, -1
    - float like 2.0, -1.7     (floats **must** contain a decimal point)
The Update Command: Semantics

- QueryEngine::update invoked to perform the specified update for any records matching the selection conditions

```c
ReturnCode update(const char* relName,
                  const RelationAttribute* left,
                  const AttributeOrValue* right,
                  int nConditions, const Condition conditions[]);
```

- Again, must verify that it typechecks wrt schema, and update indices too

- Like delete, looks a lot like a selection query, but with an update operation on top...
Query Engine API

- (cf. the Doxygen docs...)
How to Implement an Execution Engine?

• You are free to do this any way you like!
  – Modulo a few requirements we'll mention in a bit...

• We'll give sample code and interfaces to give you a starting point
  – You are free to use these, or ignore them in favor of your own design

• Provided:
  – IQueryOperator, a generic interface for operators in the tree
  – Implementations of two operators: FileScanOperator and ProjectionOperator
  – Some skeleton code in QueryEngine showing how to construct a plan
How to Implement an Execution Engine?

\[
\pi_{\text{sname}} \\
\sigma_{\text{bid}=100 \ \text{AND} \ \text{rating}>5} \\
\&_{\text{sid}=\text{sid}} \\
\text{index nested loops} \\
\text{file scan} \quad \text{index scan} \\
\text{Reserves} \quad \text{Sailors} \\
\]
How to Implement an Execution Engine?

Physical query plan

C++ implementation
IQueryOperator: an Abstract Interface

• What is an "abstract interface" in C++?

• A base class with only pure virtual methods
  
  ```cpp
  virtual ReturnCode getNextRecord(char* data) = 0;
  ```

• Other classes inherit from this base class ("implement" the interface) and fill in the method implementations

• One technical exception: virtual destructor must have implementation, but can be empty
  
  ```cpp
  virtual ~IQueryOperator() {};
  ```
What's a Virtual Method?

• C++ versus Java: in Java, **all** methods are virtual!

```cpp
class A {
  void foo() { printf("A foo"); }
  virtual void bar() { printf("A bar"); }
}
class B : A {
  void foo() { printf("B foo"); }
  virtual void bar() { printf("B bar"); }
}

void biz(A* a, B* b) {
  A* a, B* b; A* c = (A*)b;
  a->foo();
  a->bar();
  b->foo();
  b->bar();
  c->foo();
  c->bar();
}
```

**OUTPUT of call to biz(a,b):**

A foo
A bar
B foo
B bar

**QUESTION:** why declare destructors virtual?
Why Use Interfaces in the Execution Engine?

- An operator shouldn't have to know about all the different physical operators that might be below it in the tree!
IQueryOperator

• (cf. the Doxygen docs...)
Join Operator Implementation

- You will be required to implement at least two join algorithms, **nested loops join** and **index nested loops join**

- **TIP**: these can be implemented using just one join operator (that doesn't even know which join it's implementing)!

![Diagram of Nested Loops Join and Index Nested Loops Join](image)
WARNING: IQueryOperator is Not Complete!

• You will probably need to tweak IQueryOperator to suit your needs, or introduce new interfaces extending IQueryOperator will special capabilities
  – e.g., index nested loops join needs to communicate KEYS to right child, via something more than getNextTuple() alone... something like:
    
    openScan(condition); getNextTuple(); closeScan()

• Could, e.g., introduce an IScanOperator extending (inheriting from) IQueryOperator to add such methods
  – to be implemented by BOTH FileScanOperator, AND IndexScanOperator --- so that JoinOperator does not need to care about whether an index is present or not!
TIP: Make Operators for Delete and Update, Too!

• Will again require a tweak to IQueryOperator, as RecordIDs need to be returning along with records: e.g., change

```cpp
virtual ReturnCode getNextRecord(char* data) = 0;
```
to

```cpp
virtual ReturnCode getNextRecord(Record* record) = 0;
```

• What should DeleteOperator or UpdateOperator return for getNextRecord()? What should their schemas be?
  – Doesn't really matter... the results won't be printed

• What is the RecordID of the result of a join?
  – Doesn't really matter... there won't be any operators above the join that care about the RecordID (update operations don't use joins)
TIP: a Selection Operator Simplification

• Using several operators instead of one doesn't necessarily cost much in terms of efficiency

• Don't worry about implementing a selection operator that can take conjunctions of conditions

  e.g., instead of

  \[ \sigma_{\text{bid}=100 \ \text{AND} \ \text{rating}>5} \]

  just do

  \[ \sigma_{\text{bid}=100} \]

  \[ \sigma_{\text{rating}>5} \]
"Canonical" Execution Plans

• **Requirement**: however you build your execution engine, will need to produce plans that work (a) with indices, using Index Nested Loops Join, and (b) without indices (Nested Loops Join)

• Given a `select-from-where` query, will only be required to build a "canonical" execution plan of your own design
  
  – A plan fully determined by the order of relations in the "from" clause and availability of indices

  – Writeup should describe what kinds of plans you produce
Extra Credit Opportunities

• Small amount of extra credit: figure out some **heuristics** and use them to produce more efficient plans for queries/updates
  – Must document what you've done in the writeup

• Small amount of extra credit: exceptionally clean architecture and code

• Large amount of extra credit: implement a full-blown System R style query optimizer
  – Variable amount of credit, will depend on how much you do
  – Should consider different join orders and do some sort of cost estimation to compare plans
  – Will require maintaining and using **cardinality statistics**, at a minimum