MercuryDB

Main Memory Single-Schema DB Generator

Doug Ilijev, Cole Stewart
Outline

• Why MercuryDB?
• Modules
• The API
• Query and Join Optimizations
• Building the Database
• Target Module Generator
Outline

• Why MercuryDB?
• Modules
• The API
• Query and Join Optimizations
• Building the Database
• Target Module Generator
Motivation

• Most relational database systems
  o have 1 or more schemas that can also change
  o are bottlenecked by I/O in queries of small complexity
  o provide a layer of abstraction between the client and the data

• Would be nice to have an API that is schema-dependent
  o Interesting opportunities for optimization
The Answer: MercuryDB (for Java)

- Schema is generated in Java source code
- Generated code exposes an API to the client
- All queries are done using anonymous Iterators
  - no stream objects are buffered while processing queries
    - (except for hash joins)
  - Backend implementation is very “functional”
- 1-User, Non-Persistent Database
The Answer: MercuryDB (for Java)

• The generated database is compiled with the source program
• 4 steps for use:
  o Compile target package
  o Run MercuryDB on target to create a schema
  o Add Mercury hooks to original program
  o Client application uses the Mercury Schema to query facts about the target
Outline

• Why MercuryDB?
• Modules
• The API
• Query and Join Optimizations
• Building the Database
• Target Module Generator
Modules

Target (.class files) → mercurydb
[targtPackage]
[schemaPackage] → Mercury Schema

Add Database Hooks
(Target → Target')

Target' → Client Application
+ Query Logic
Control Flow

<table>
<thead>
<tr>
<th>Client</th>
<th>Target’</th>
<th>MercurySchema</th>
</tr>
</thead>
<tbody>
<tr>
<td>entry:main()</td>
<td>(1) exercise Target’ API</td>
<td>(2) updates to Mercury DB</td>
</tr>
<tr>
<td></td>
<td>(3) query state of DB</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(4) examine state of DB</td>
<td></td>
</tr>
</tbody>
</table>

Diagram:
- Entry:main() initiates the process.
- Exercise Target’ API
- DB state query
- DB state examination
- Update to Mercury DB
Example Input

- Possible input package to javadb.Main
Example Output

- 1 output table per input class

*public operations omitted for brevity
Resulting Code

Bring it all together

Generated
Outline

• Why MercuryDB?
• Modules
• The API
• Query and Join Optimizations
• Building the Database
• Target Module Generator
Filter Plumbing Diagram

\[ \text{SELECT} \ast \text{ FROM EMP WHERE ID=5} \]

\begin{verbatim}
EmpTable.scan()
    .filter(EmpTable.fieldId(), 5)
\end{verbatim}
Filter Plumbing Diagram

SELECT * FROM EMP WHERE ID=5 AND NAME="Don"

EmpTable.scan()
    .filter(EmpTable.fieldId(), 5)
    .filter(EmpTable.fieldName(), "Don");
Filter - Value Unions

• Retrieve objects where a field could be a set of values
• `filter(FieldExtractable fe, Object... values)`
  o accepts any number of values, validates with `.equals()`

```java
select * from emp
where id=5 and (name="Don" or name="Scarlett");
->
EmpTable.scan()
  .filter(Emp.fieldId(), 5)
  .filter(Emp.fieldName(), "Don", "Scarlett");
```
Filter - General Plumbing Diagram

```sql
select * from emp,
where and loc="Austin" and
   (name="Don" or name="Scarlett");
```

Stream containing column Emp

```java
Emp.
    .filter(Emp.fieldLoc(),"Austin")
    .filter(Emp.fieldName(),"Don","Scarlett");
```
Indexes

- Must use @Index attribute for fields you want indexed in the db

```
Input Class to javadb.Create

Class Emp {
    @Index
    public int id;
    public String name;
    ...
}
```

- Utilized in queries and joins

```
Output Table

Class EmpTable {
    //standard table
    static Set<Emp> table
    
    //generated index, values are also in table
    static Map<Integer, Emp> idIndex = ...
    ...
}
```
Query Plumbing Diagram

**SELECT * FROM EMP WHERE ID=5**

before: -> EmpTable.scan().filter(EmpTable.fieldId(), 5)
   now:   -> EmpTable.queryId(EmpTable.fieldId(), 5)

*No Index*

![Diagram showing the plumbing of a query without an index.]

*With Index on Emp.id*

![Diagram showing the plumbing of a query with an index.]

IndexRetrieval (EmpTable.fieldId(), 5)
Query Plumbing Diagram

```
SELECT * FROM EMP WHERE ID=5 AND NAME="Don"
-> EmpTable.queryIdName(5, "Don")
```

No Index

```
Retrieval(EmpTable.scan()) -> Filter(Emp.id="Don") -> Filter(Emp.name="Don")
```

With Index on Emp.id

```
IndexRetrieval (EmpTable.fieldId(), 5) -> Filter(Emp.name="Don")
```
Query Methods

• Queries can query one or more fields in a Class

```java
Class EmpTable {  // Has fields Emp.Id and Emp.name
    queryId(int id)  // All Emps where Emp.id=id
    queryName(Str name)  // All Emps where Emp.name=name
    queryIdName(int id, Str name) // All Emps where Emp.id=id
     // and Emp.name=name
}
```

• $M = || (P([\text{fields in Emp}]) \setminus \emptyset) ||$

• Aggressively uses indexes

$$M = \sum_{i=1}^{n} \binom{n}{i} = 2^n - 1$$
Query Methods

\[ M = \sum_{i=1}^{n} \binom{n}{i} = 2^n - 1 \]

• Reasonable for small \( n \)
• Consider class with 30 fields
  - \( M = 2^{30} = 1 \text{ GB} \) (assuming 1 byte per method - laughable)
• Are such unwieldy signatures practical?
  - No! Too complicated for users
  - Generate only “reasonable cardinality” methods (\( k \) fields)
• Let \( k \) be configurable
  - By default \( k=4 \)
Query Methods

• Consider $n=30$ and $k=4$

$$M = \sum_{i=1}^{k} \binom{n}{i} = \sum_{i=1}^{4} \binom{30}{i} = 31930$$

• Still huge, but given enough memory might at least be possible
Query Methods

• What about all the methods not generated?
  o “I want to restrict results on more fields!”
• You still can, by chaining the queries with filters

```java
queryABCD(a, b, c, d)
    .filter(onFieldE, eVal)
    .filter(onFieldF, fVal) ... ; // pseudocode
```

• The query call is optimized over those fields (using best index)
• Filters continue to filter on additional fields
  o Up to the client to know which fields are less likely candidates for optimal index retrieval
Query Methods - Future Work

• Problems with the current scheme
  o Too many methods
  o Complicated signatures
  o Limited number of fields

• Be more general
  o We want to create a single query(...) method
  o Deduce which fields from the params, like filter(...)
  o Less generated code
  o Less cognitive load for the programmer
Joins

- Filters can be applied before or after the join operation

```java
join(EmpTable.queryId(5).joinOn(EmpTable.fieldId()), OrderTable.joinOnFieldEid())
```

```java
join(
    EmpTable
    .queryId(5)
    .joinOn(EmpTable.fieldId()),
    OrderTable.joinOnFieldEid()
)
```

```java
join(
    EmpTable.joinOnFieldId(),
    OrderTable.joinOnFieldEid()
).filter(EmpTable.fieldId(), 5)
```

`select * from Emp, Order where Emp.id=5 and Emp.id=Order.id`
Join Plumbing Diagram

\[
\text{select } * \text{ from Emp,Order where Emp.id=5 and Emp.id=Order.id}
\]

```
join(
    EmpTable
    .queryId(5) // id not indexed
    .joinOn(EmpTable.fieldId()),
    OrderTable.joinOnFieldEid()
)
```
Join Plumbing Diagram

```
select * from Emp, Order where Emp.id=5 and Emp.id=Order.id
```

```
Join(Emp.fieldId(), Order.fieldId())
joinOn(EmpTable.fieldId())
joinOn(OrderTable.fieldId())
join(EmpTable
    .joinOn(EmpTable.fieldId(), OrderTable.joinOnFieldEid())
    .filter(EmpTable.fieldId(), 5)
).filter(EmpTable.fieldId(), 5)
```
**Join Plumbing Diagram**

```
select * from Emp,Order where Emp.id=5 and Emp.id=Order.id
```

```
IndexRetrieval(Emp.fieldId(), 5)

Retrieval(OrderTable.scan())

Join(Emp.fieldId(), Order.fieldId())

join(EmpTable
    .queryId(5) // id is indexed
    .joinOn(EmpTable.fieldId(), OrderTable.joinOnFieldId())
)
```
Joins

- Can join on any field of a class instance, including the instance itself
- Join operations take one or more equality predicates
  - Each predicate holds two streams
  - Only equality join relations are currently supported
  - Could also use output of a filter or retrieval as input
    - everything is a stream!

\[
\text{select } * \text{ from } A, B \text{ where } A.x=B.y \rightarrow \text{JoinDriver.join(} ATable.joinOnFieldX(), BTable.joinOnFieldY() \text{)}
\]
Join Results

- Streams typically return table elements
  - Stream<A>, Stream<B>, etc.
- Still a stream, but a stream of what?
  - JoinRecord
- A JoinRecord is essentially an alias for Map<Class<?>, Object>
  - Values are always instances of the type defined in the key

```java
for (JoinResult jr : JoinDriver.join(
    ATable.joinOnFieldX(),
    BTable.joinOnFieldY()).elements()) {
    A a = (A)jr.get(A.class);
    B b = (B)jr.get(B.class);
}
```
Joins (Use Case)

```java
public class Order {
    @Index
    public int ono;
    public Customer cno;
    ...
}
```

```
public class Odetail {
    @Index
    public int ono;
    public Customer cno;
    ...
}
```

```sql
select * from Order, Odetail where Order.ono=Odetail.ono;
```

```java
JoinDriver.join(
    OrderTable.joinOn(OrderTable.fieldOno()),
    OdetailTable.joinOn(OdetailTable.fieldOno()));
```

```java
select * from Order, Odetail where Order.ono=Odetail.ono and Order.ono=5
```

```java
JoinDriver.join(
    OrderTable.queryOno(5).joinOn(OrderTable.fieldOno()),
    OdetailTable.joinOn(OdetailTable.fieldOno()));
```
Joins (Use Case)

```
select * from Order, Odetail
where Order=Odetail.ono;

JoinDriver.join(
    OrderTable.joinOn(OrderTable.itself()),
    OdetailTable.joinOn(OdetailTable.fieldOno());

select * from Order, Odetail
where Order=Odetail.ono and Order.ono=5

JoinDriver.join(
    OrderTable.queryOno(5)
        .joinOn(OrderTable.itself())
    OdetailTable.joinOn(OdetailTable.fieldOno());
```

```
public class Order {
    @Index
    public int ono;
    public Customer cno;
}

-- more likely

public class Odetail {
    @Index
    public Order ono;  -- more likely
    public int qty;
}
```
Outline

• Why MercuryDB?
• Modules
• The API
• Query and Join Optimizations
• Building the Database
• Target Module Generator
Query Optimizations

• Determine the best code to execute depending on indexes etc.
  o No fields have an index?
    ▪ Apply a filter
  o Does only one of the fields have an index?
    ▪ Use the index
  o Do multiple fields have an indexes?
    ▪ Use the one for table with smallest cardinality

• Main memory database
  o Don’t worry about disk I/O latency or locality
  o Random memory access for reading tuples means indexes should ALWAYS be used
Single Predicate Join Algorithm

• Given two streams, there are 3 primary cases
  o **Join fields of both streams are indexed**
    ▪ Do index intersection
  o **Only one stream is indexed**
    ▪ Scan over non-indexed stream, getting join columns using other stream’s index
  o **Neither stream is indexed**
    ▪ Use hash join
    ▪ Hash the stream with smaller cardinality
Sideways Information Passing

• Def. — sending information from one query operator to another in a fashion not specified by the query evaluation tree
• Index information is passed through to join operations
• Join algorithm makes a runtime decision on whether or not to use a field’s index
Multi-Predicate Join Algorithm

• Users can also specify their own join execution plans
  o Could have bushy joins
  o Or right-deep or left-deep joins
• Everything is a stream
  o Could add support for other join strategies
• Current default join strategy follows System R strategy
Multi-Predicate Join Optimization

```java
// select * from order, odetail, emp
// where order=odetail.ono and order.eno=emp.eno;
// JoinDriver will join the predicates like System R
JoinDriver.join(
    new JoinPredicate(
        OrderTable.joinOnItself()),
    new JoinPredicate(
        OrderTable.joinOnFieldOno(),
        OdetialTable.joinOnFieldOno()),
    new JoinPredicate(
        OrderTable.joinOnFieldEno(),
        EmpTable.joinOnItself())));
```
Outline

• Why MercuryDB?
• Modules
• The API
• Query and Join Optimizations
• Building the Database
• Target Module Generator
Constraints on Source Package

• Only public fields are included in database
  o Could use reflection to retrieve private fields, but this is slow and cumbersome
• To index a field, must use `@Index` annotation
• Must not use class names that conflict with those created by the database creator
• Must maintain consistency in the database by using update calls defined by the tables
  o Bytecode modification makes this simple!
Javassist Bytecode Generation

- Javassist is a class library for modifying bytecodes in Java
- Allows us to insert hooks in the client code that update the db
  - injects table insert operation at the end of each constructor
    - ex. EmpTable.insert(this) in class Emp
  - injects a table update method at the end of every setter method for fields that are indexed

```java
Class Emp {
    @Index
    int id;
    String name;

    public void setId(int id) {
        this.id=id; }
}
```

```java
Class Emp {
    @Index
    int id;
    String name;
    //Note: type sig could be anything
    public void setId(int id) {
        this.id=id; EmpTable.setId(this, id); }
}
```
Table Field Declarations

• Every generated table class has a set whose elements are weakly referenced to store all instances of its mapped class
  o public static Set<WeakReference<Foo>> table;

• Since only id and name are indexed, two maps are generated
  o private static Map<Integer, Set<WeakReference<Foo>>> idIndex;
  o private static Map<String, Set<WeakReference<Foo>>> nameIndex;
Database Consistency

• How are objects added to the database?
  o via bytecode modification

• How are indices kept consistent?
  o must use setters in database

• What happens when instances in the table go out of scope?
  o Garbage Collected (deleted)
  o Tables hold WeakReferences
  o Garbage Collector ignores the references in tables
Outline

• Why MercuryDB?
• Modules
• The API
• Query and Join Optimizations
• Building the Database
• Target Module Generator
Mercury Retrograde

Single-Schema Database
Target Module Generator
(for MercuryDB)
JavaDatabase implements IDatabase

• Converts MDB SQL scripts to Java classes
• Modified MDB Project
• Same Parser
• New IDatabase implementation for JavaDB
• Mustache to format output
public interface IDatabase {

    // target module class defs
    public void create(...);
    public void index(...);

    // client module actions
    public void insert(...);
    public void update(...);
    public void delete(...);
    public void selectAll(...);
    public void select(...);

    // format templates
    public void commit();

    // reset data
    public void abort();

    // do nothing
    public void close();

}
JavaDatabase - Target Module

• `create ...`
  o 1-to-1 target package class definition

• `index ...`
  o add `@Index` annotations to above class definitions
public class {{tableName}} {  
  {{#fields}}
  {{#isIndexed}}
  @Index
  {{/isIndexed}}{{#isInteger}}
  int {{name}};
  {{/isInteger}}{{#isString}}
  String {{name}};
  {{/isString}}{{/fields}}
  public {{tableName}}(  
    {{#fields}}
    int {{name}},
    {{/isInteger}}
    String {{name}},
    {{/isString}}
  )  
  {
    {{#fields}}
    this.{{name}} = {{name}};
    {{/fields}}
  }
}
create table emp (  
  empno int,  
  age int  
);  
index emp.age;  
commit;

class EmpTable {  
  int empno;  
  @Index  
  public int age;  
  public EmpTable(  
    int empno,  
    int age  
  )  
  {  
    this.empno = empno;  
    this.age = age;  
  }  
}
JavaDatabase - Client Module

- **insert ...**;
  - insert a record into the table (e.g. ATable)
  - create new instance of class (of e.g. ATable) corresponding to table
  - insert into an ArrayList to keep reference live

- **update ...**;
  - use setters to update objects

- **delete ...**;
  - delete matching objects from ArrayList
JavaDatabase - Client Module

- `select ...;`
  - Return references to entire tuples
    - `select [fields] = select *`
    - simplifies code generation somewhat
  - Uses API generated by JavaDB
**JavaDatabase - Target Code**

```java
insert into emp values (1, 25);
insert into emp values (2, 47);
.
update emp set age=26 where empno=1;
.
select emp where empno=1;
.
delete emp where empno=2;
.
ArrayList<Emp> emps = new ...
emps.insert(new Emp(1, 25));
emps.insert(new Emp(2, 47));
...
... i = EmpTable.queryEmpno(1);
for (Emp e : i.elements()) // update
    e.setAge(26);
i = EmpTable.queryEmpno(1);
for (Emp e : i.elements()) // select
    ...
    // do something
i = EmpTable.queryEmpno(2);
for (Emp e : i.elements()) // delete
    emps.remove(e);
```
Outline

• Why JavaDB?
• Modules
• The API
• Query and Join Optimizations
• Building the Database
• Target Module Generator
References

[1] Zachary G. Ives and Nicholas E. Taylor,
   http://repository.upenn.edu/cgi/viewcontent.cgi?article=1045&context=db_research

   “Access Path Selection in a Relational Database System,” in ACM, 1979
   http://dl.acm.org/citation.cfm?doid=582095.582099
Questions?
[Backup Slides]
JavaDatabase - Client Mustache
JavaDatabase - Client Module
What the Target Doesn’t Know

• Q: If the authors of the target module don’t know about JavaDB, how do @Index annotations get added?

• A: We can allow the author of the client to specify which fields to add Indexes to, and use reflection to add that information to the generated API.
[Removed Slides]
Client API Introduction

• Every public field has a corresponding query method in the table
• Every query method returns an instance of the type its table contains
• Some fields can be indexed
  o generated query methods will utilize these where possible
  o index is essentially a Map<FieldVarType,ClassVarType>
• Queries can be chained and filtered together
The Steps (old version)

Source Package

```
javadb.Main
[sourcePackage]
$dbOutPackage$
```

Source Package + Database Package

Add Database Operations to Source Package
Multi-Predicate Join Optimization

• Given a join operation with multiple predicates, in what order are they processed?
  o Look at System R

• Look at JoinPredicate’s compareTo method to see the order in which joins are processed

```java
/**
 * 1. Number of indices (greater first, i.e. descending)
 * 2. Cardinality of its streams (smaller first, i.e. ascending [natural])
 */

@Override
public int compareTo(JoinPredicate o) { … }
```
Control Flow

Compiled Source Package ➔ javadb.Main [sourcePackage] [dbOutPackage] ➔ Database Package

Source Package* + Database Hooks ➔ Add Database Hooks to Source Package

Client Application (Query Logic)

* may or may not be modified from the original source package depending on implementation
Code Template
Self Joins

• select * from A as A1, A as A2 where A1.x=A2.y
• Not currently possible in JavaDB
  o Aliases are not possible
• A solution would be interesting...