MercuryDB

Main Memory Single-Schema DB Generator

Doug Ilijev, Cole Stewart

Open by mentioning that this is work we are actively working on

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

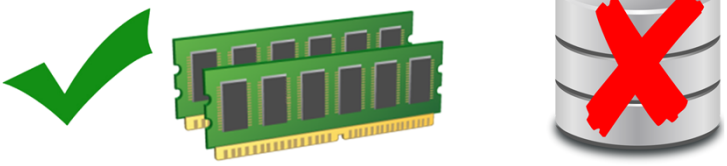
- Would be nice to have an API that is schema-dependent
  - Interesting opportunities for optimization

Most relational database systems are part of a client-server architecture, we don't necessarily need that here
Since the entire database lives in main memory along with the client, we don't need too much of a client-data abstraction layer.

Since there's only 1 schema, this will also allow us to generate a schema-dependent API that integrates well with
The data the database holds.

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 streams are buffered while processing queries
    - (except for hash joins)
  - Backend implementation is very “functional”
- 1-User, Non-Persistent Database



The schema in MercuryDB is simply a set of Java classes that are generated.
The generated code exposes an API to the client, allowing the client to perform query operations on the database.

Since everything is in main memory already, we want to keep from using more memory where we can, so
Query operations do not buffer streams while processing queries.

It is more of the iterator-driver “push-style” approach.

There is only 1 user here (for now) as well, so we don’t need to worry about concurrency and transactions (yet).

The Answer: MercuryDB (for Java)

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

The main point here is that since the database is schema dependent,
We shouldn't have to generate the schema too often. Still though, it is very fast.

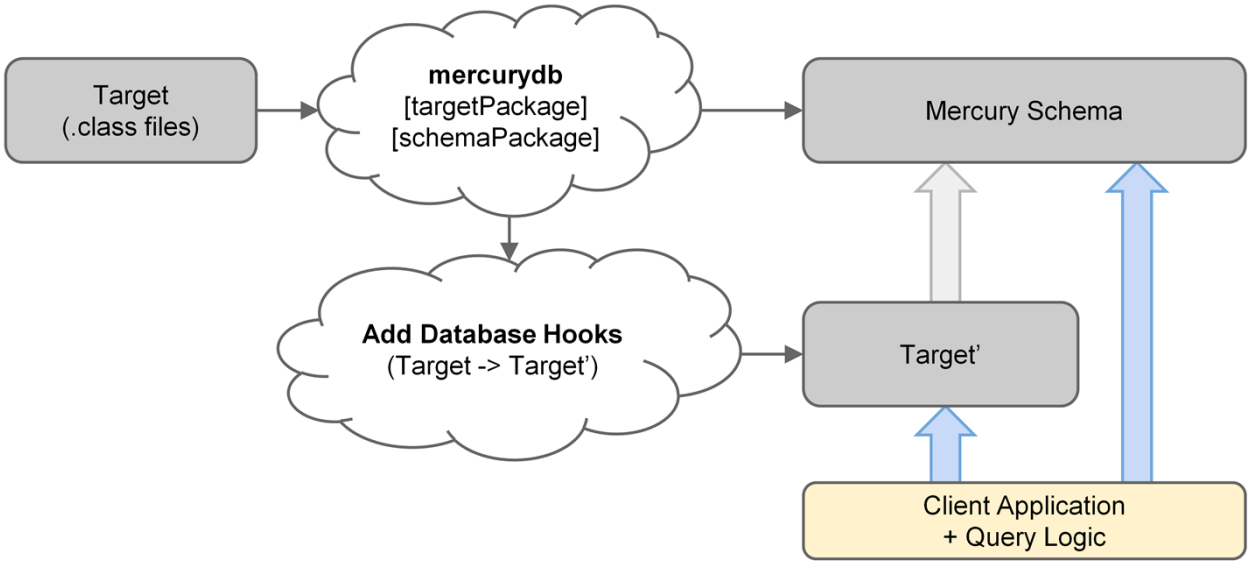
There are 4 main steps to use MercuryDB (compile runmercury addhooks querydatabase)

So you can't write queries until the generated schema is connected to the client.

Outline

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

Modules



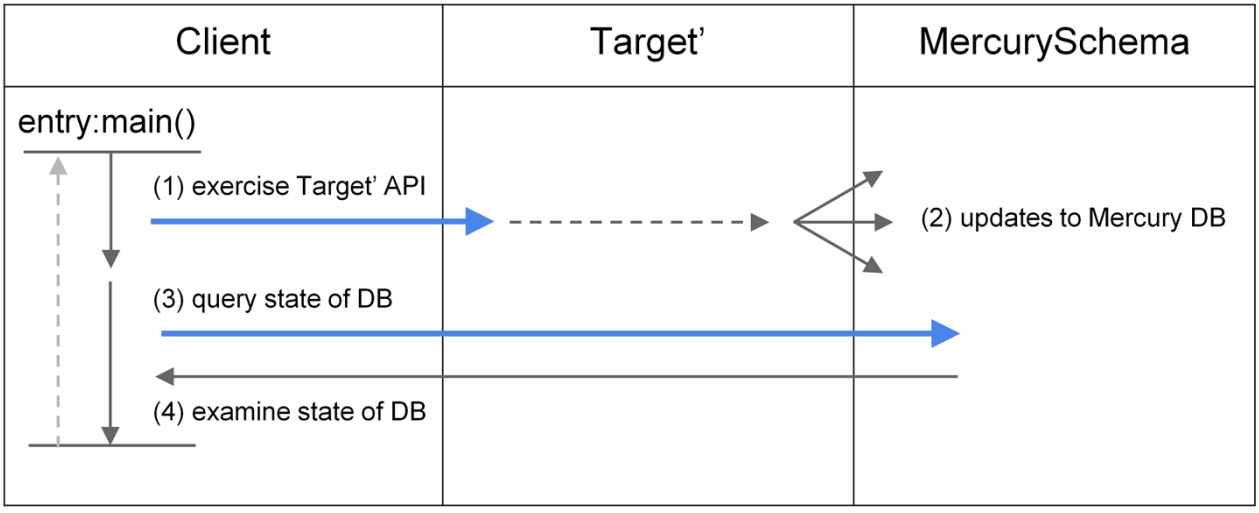
So MercuryDB's main goal here is to generate a schema.

After given a set of class files, it will generate a schema (.java files) and add hooks into the input code (original set of .class files) to keep the database consistent (more on this later).

So now, Target essentially becomes a modified target. The source code for target is not completely representative of its compiled class files at this point.

Now, the Client application can use query methods defined by Mercury to query the target.

Control Flow



After the schema has been created and the client is running, this is the control flow

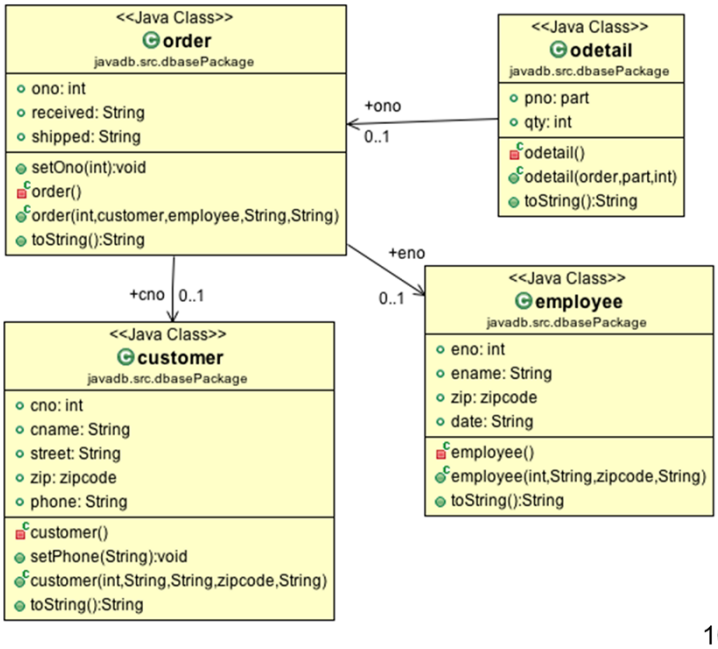
The client will exercise Target' API – creating objects, setting fields, etc.
This will spawn updates in the database

Then the client will query the state of the database
To which mercury will reply with the result

And, of course, this loop runs forever until the program ends.

Example Input

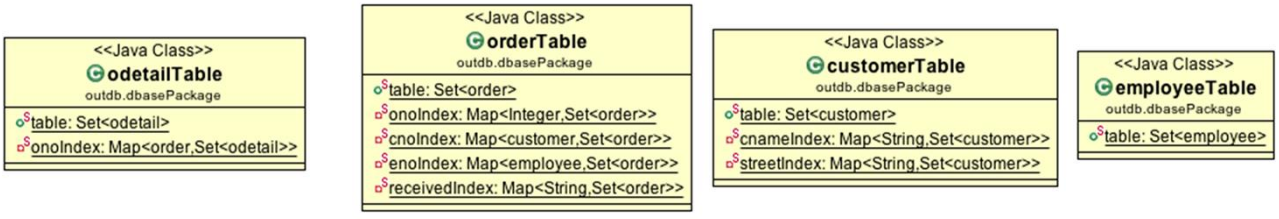
- Possible input package to **mercurydb**



Here is an example of what an input package might look like.
These classes are modelled like something that you would typically see in a database, but MercuryDB could handle any input package that abides by its (small) set of constraints.

Example Output

- 1 output table per input class



*public operations omitted for brevity

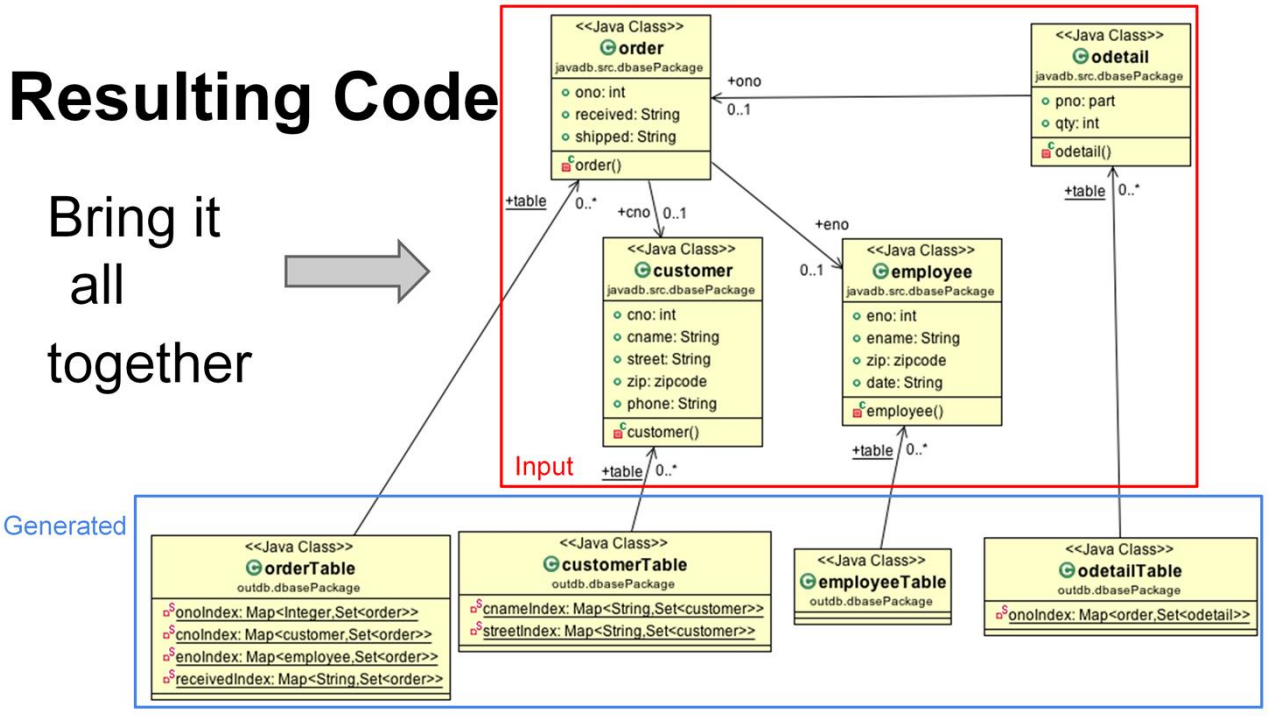
Here is what MercuryDB will produce when given the previous package as input.
Note that there is a 1-1 mapping from classes to tables.

Order - OrderTable
Odetail - OdetailTable
Customer - CustomerTable
Employee - EmployeeTable

The public operations/methods have been omitted here,
but you can still see the table and data structures present that contain the data
in the database.

Resulting Code

Bring it all together



This is what the input classes look like and how the generated schema integrates with them.

The arrow between each table class and its corresponding container class represents the tables of container class objects themselves.

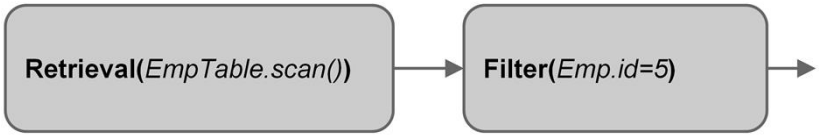
Both generated schema and input code work together.

Outline

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

Filter Plumbing Diagram

```
SELECT * FROM emp WHERE id=5
```




```
EmpTable.scan()
    .filter(EmpTable.fieldId(), 5)
```

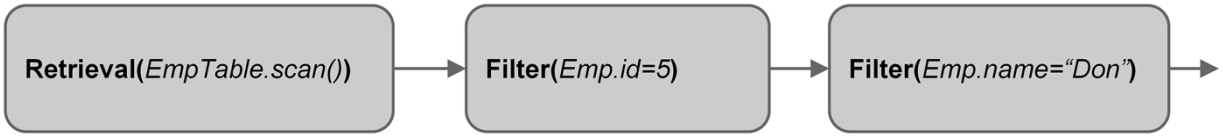

We'll start by explaining two key pieces of the API:

Table Scan - returns a stream of the objects it contains i.e. EmpTable.scan() returns Stream<Emp>

Filter methods - filters a stream of some object such that one of its fields is some value, in this case Emp.id=5

Filter Plumbing Diagram

SELECT * FROM emp WHERE id=5 AND name="Don"




```
EmpTable.scan()
    .filter(EmpTable.fieldId(), 5)
    .filter(EmpTable.fieldName(), "Don");
```


Since filters accept a Stream and return a Stream,
Filters can be chained together.

So we can make something like the query here using only filters.

Notice in the API how the field is specified.
The schema exposes these methods that will allow the filter function to extract the field "id" or "name" from a given emp object.
It is not done by reflection, so it is very fast.

Filter - Value Unions

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

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

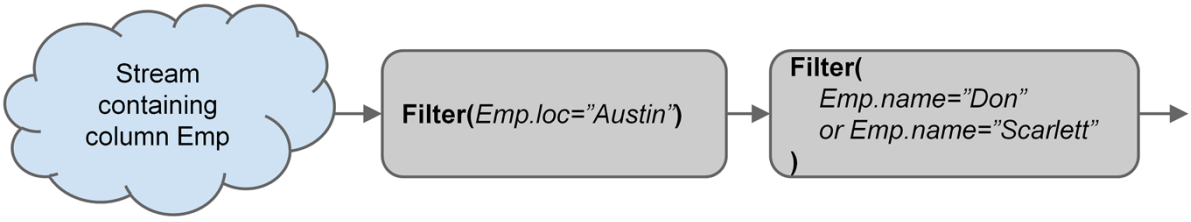

Based on the examples you've seen so far it might appear that if you want to filter objects matching a set of values, you might have to build these diagrams more than once, but our API
supports filtering on a set of values.

To use the filter method you simply have to specify the field you want to filter,
and then you can specify any number of values for that field, and we will return objects that match any of those values.

A hypothetical SQL query with value unions might look like this, and would translate to the following code in our API.

Filter - General Plumbing Diagram

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




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

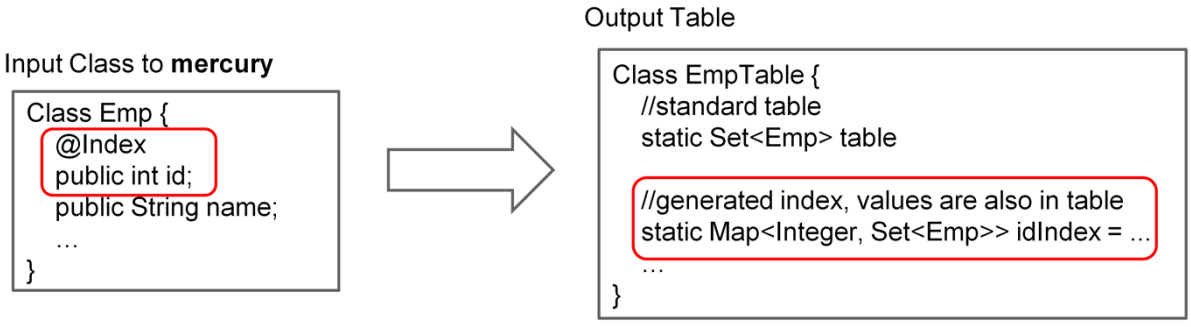

This slide is mainly to show two things:
(1) filters can be applied after any kind of operation
that produces a stream that contains the type you are trying to filter.

(2) Filters use Java varargs to implement a union operation on field values in the predicate.

If you place a filter in the query over a type that the stream does not contain, we throw a helpful error,
so that the user knows what they did did not make any sense.

Indexes

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



- Utilized in queries and joins

At this point, it is important to note that MercuryDB supports indexes.
The client can specify the index by using the @Index annotation in the input source package.
There is a 1-1 mapping of indexes to index structures in the database.

Indices are implemented by Maps
key=type of field
value= type of Object the table contains

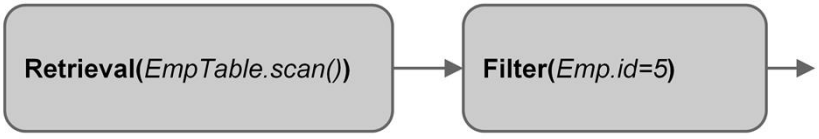
So if there is an index on Emp.id, there will be an idIndex in the EmpTable class.

These table indices are not intended to be used directly. They are used in the Tables' query methods describe next.

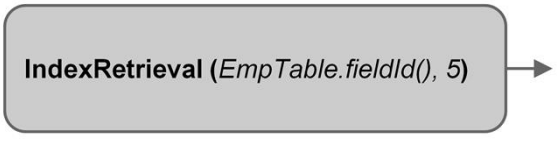
Query Plumbing Diagram

```
SELECT * FROM EMP WHERE ID=5
before: -> EmpTable.scan().filter(EmpTable.fieldId(), 5)
   now: -> EmpTable.queryId(5)
```

No Index



With Index on Emp.id



Using the Tables' query methods provide an API that exposes an easy way to retrieve Object instances from the database where one or more of its fields are equal to some value.

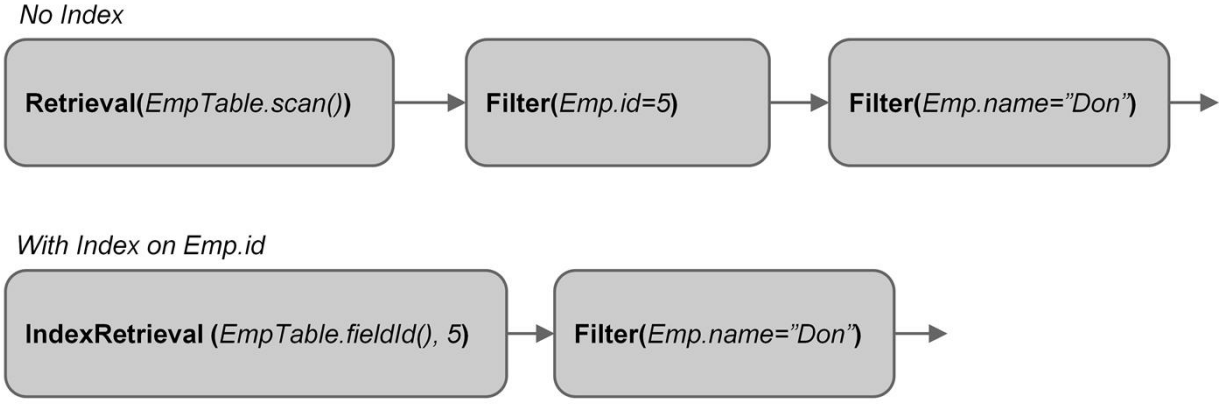
There are two cases:

Field is not indexed -> do same as before: scan and filter

Field is indexed -> do an index retrieval, we already have the set that matches the constraints of the predicate.

Query Plumbing Diagram

```
SELECT * FROM emp WHERE id=5 AND name="Don"
-> EmpTable.queryIdName(5, "Don")
```



This is the same as before, but we introduce the idea of querying multiple fields of a class in one method.

When using a multi-field query method,
 the index with the smallest set of objects that matches the constraints of the predicate is used to seed the final plumbing diagram.
 then filters are applied to retrieve the final result

Query Methods

- Queries can query one or more fields in a Class

```
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([fields in Emp]) \ Ø) ||
- Aggressively uses indexes

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

M is the number of query methods
n is the number of fields

We don't produce a method for querying on 0 fields (the idea of returning everything is already taken care of with scan()), so we subtract the empty set from the power set, and end up with 2^n - 1 total methods.

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 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

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

This slide is mainly to show how the # of query methods can grow if we continue to add an exponential number of methods for each successive # of fields.

Since query methods will likely be parameterized soon, this slide only shows that we thought about the problem.

Still, this is how it is done currently with 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

Again, for k=4, this is *OK*, but if the query methods were parameterized everything would be ok.

Query Methods

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

```
XTable.queryABCD(a, b, c, d)
    .filter(XTable.fieldE(), eVal)
    .filter(XTable.fieldF(), fVal) ... ;// pseudocode
```

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

Since query methods ultimately reduce to an index retrieval and a series of filters, The user is free to apply filters in the same fashion after performing a query.

Ideally, the query method will do the optimum series of calls to retrieve data from the database.

Query Methods - Future Work

- Problems with the current scheme
  - Too many methods
  - Complicated signatures
  - Limited number of fields
- Be more general
  - We want to create a single query(...) method
  - Deduce which fields from the params, like filter(...)
  - Less generated code
  - Less cognitive load for the programmer

Why didn't we do this already?

Once you start generating code, it's hard to know when to stop.
Early versions of the type system in the generated code made this very difficult to realize.
Now the type system is much more consistent, and better for this kind of refactoring.

Coming soon :)

Joins

- Filters can be applied before or after the join operation

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

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

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

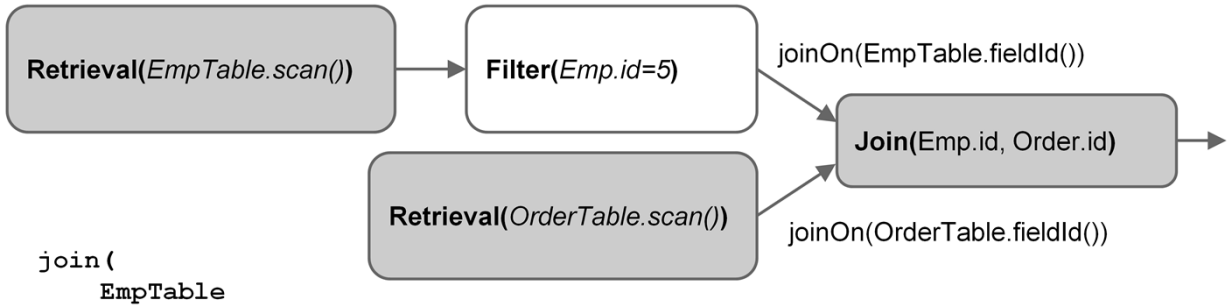
Produce the same result with different plumbing diagrams.

Here we introduce the notion of joins with our API.
Assume there is no index on id
We have a simple query with a filter operation on Id=5 and a join on Emp.id=Order.id
The API allows you to either put this filter after the join, or before the join in the form of a filter or query method.

You'll get the same result, you just have the control to get that result in different ways.

Join Plumbing Diagram

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

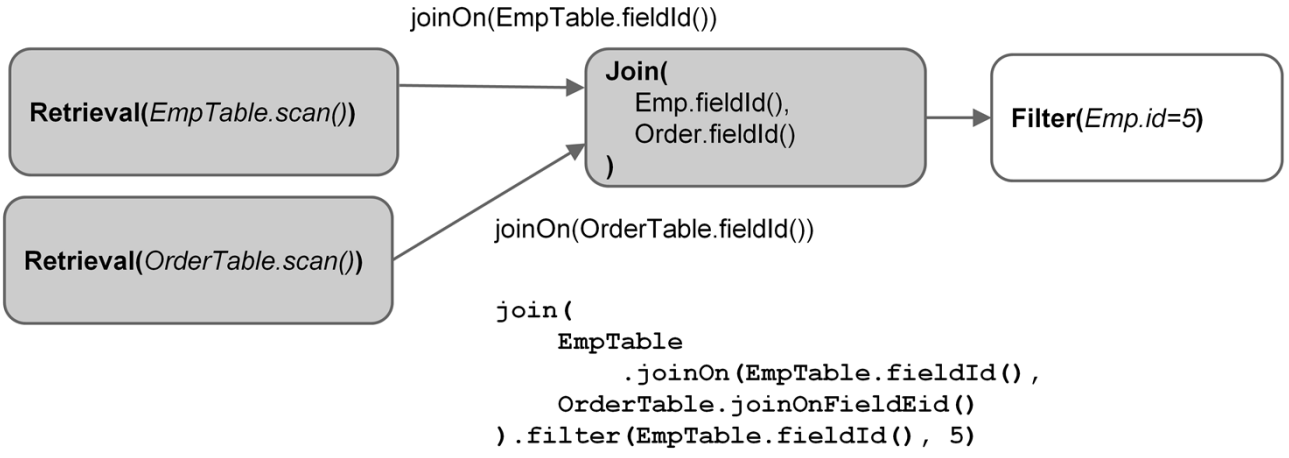



```
join(
    EmpTable
        .queryId(5) // id not indexed
        .joinOn(EmpTable.fieldId(),
    OrderTable.joinOnFieldEid()
)
```


See how the filter is done before the join here

Join Plumbing Diagram

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

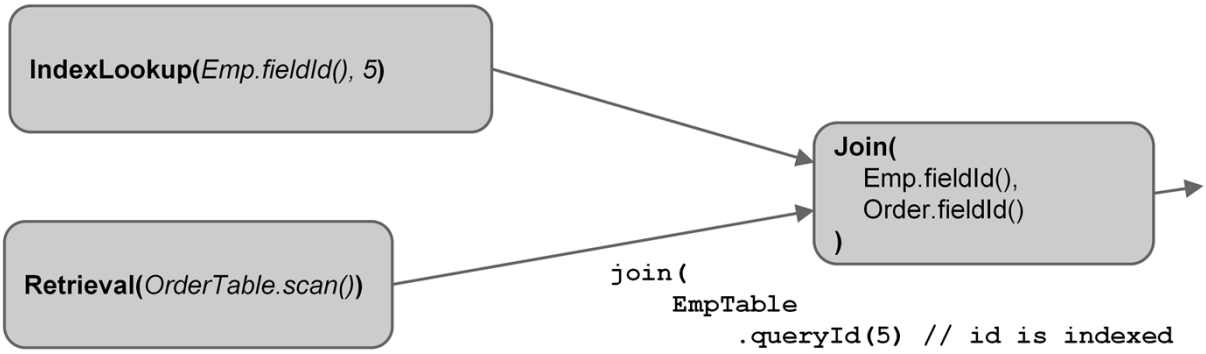



```
join(
    EmpTable
        .joinOn(EmpTable.fieldId(),
    OrderTable.joinOnFieldEid()
).filter(EmpTable.fieldId(), 5)
```


See how the filter is done after the join here.

Join Plumbing Diagram

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



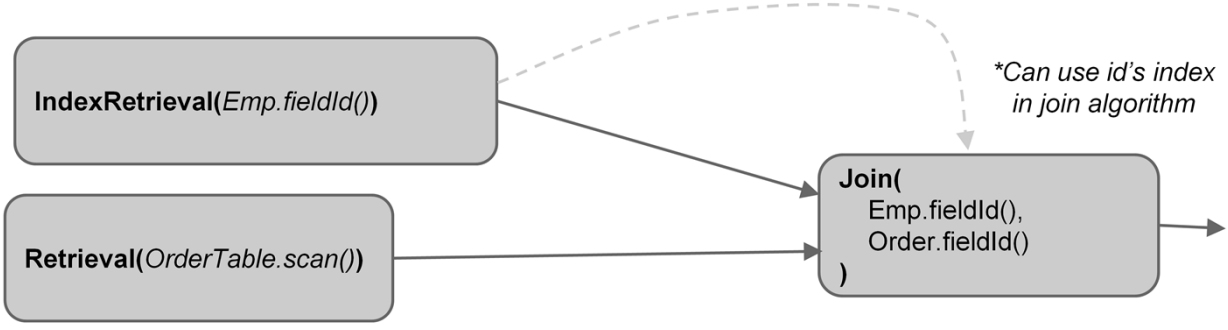

```
join(
    EmpTable
        .queryId(5) // id is indexed
        .joinOn(EmpTable.fieldId(),
    OrderTable.joinOnFieldEid()
)
```


We want to use the query method on id so that we can take advantage of the Index.

EmpTable.queryId(5) effectively reduces to an index lookup, we'd much rather do this than Apply a filter after the join operation.

Join Plumbing Diagram

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




```
join( // assume Emp.id is indexed
    // new IndexRetrieval(EmpTable.idIndex).joinOn(EmpTable.fieldId())
    EmpTable.joinOnFieldId(),
    OrderTable.joinOnFieldid()
)
```


Ideally, we'd like to use indices where we can as input to joins to take advantage of the more optimal join algorithms.

In order to use these indices we need to use the table's predefined joinOnFieldX methods to
Easily create IndexRetrieval objects.

Join operations experience a significant speedup when they can take advantage of indexes,
So if you have an index on a field you will want to try to use it in the join operation by using the joinOnField methods.

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

```
select * from A,B where A.x=B.y ->
JoinDriver.join(ATable.joinOnFieldX(),BTable.joinOnFieldY())
```

- Only equality join relations are currently supported
- Could also use output of a filter or retrieval as input
  - everything is a stream!

Note that if only one predicate is used, the argument does not have to be wrapped in a Predicate object.
Everything is a stream, but you do have to remember to set the join field

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
- Provides an easy mechanism for retrieving elements from a tuple

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


This specifies what is returned from a join operation. The result is still a Stream, but instead of containing instances of one type, it contains instances of multiple types in the form of a Map.

That is, each record in the stream is a JoinRecord, which is an alias for a Map<Class<?>, Object>

If we joined on multiple predicates consisting of the tables ATable, BTable, CTable, We would get a **Stream of Maps with the keys A.class, B.class, and C.class where the value is an instance of the key class**

Note that this current implementation prevents us from doing self joins, since we can only have one instance for each type in a record.

Joins (Use Case)

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

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

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

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

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

```
public class Odetail {
   @Index
   public int ono;
   public int qty;
   …
}
```


Simple join operation. Showing how int ono is in both classes and we want to find all Orders and Odetails where ono=ono;

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;

}
```

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

}
```


In Java, the foreign key will often be a reference to the Object itself.

the itself() idea means that we can test if
the value of a field one input of the join is equal to the value of the other input in the join

A.x=B instead of A.x=B.x

This is somewhat equivalent to the idea of **Following Links**
That is, one object has a reference to another object, and we want to find the set of the other objects that some object maps to.

But there is an important distinction.
We are dealing with two separate streams. The other stream may have be subset of those instances present in the other's field.

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.
  - No fields have an index?
    - Apply a filter
  - Does only one of the fields have an index?
    - Use the index
  - Do multiple fields have an indexes?
    - Use the one for table with smallest cardinality
- Main memory database
  - Don’t worry about disk I/O latency or locality
  - Random memory access for reading tuples means indexes should ALWAYS be used

This is for queryABC(...) methods in a table.

If there is no index,
  we only apply filters
if there is one index
  we use that index
if there are multiple indices (say A,B, and C are indexed)
we would use the index where the

Single Predicate Join Algorithm

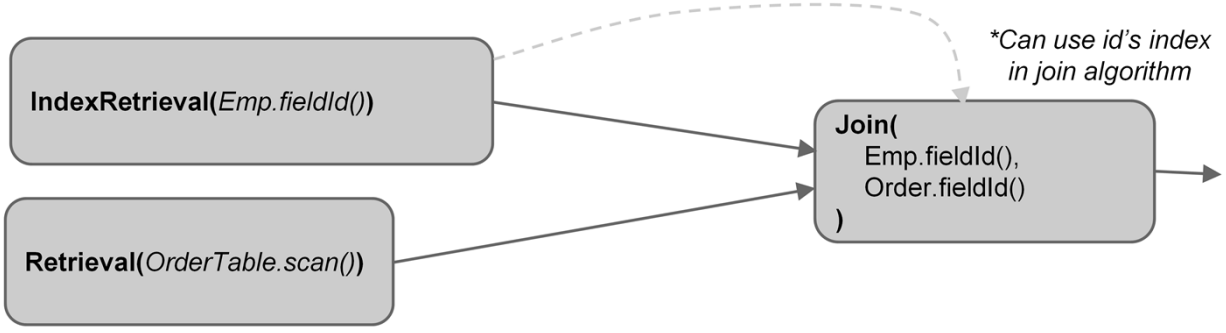
- Given two streams, there are **3** primary cases
  - **Both Inputs are IndexRetrievals**
    - Do index intersection
  - **Only one stream is an IndexRetrieval**
    - Scan over non-indexed stream, getting join columns using other stream's index
  - **Neither stream is an IndexRetrieval**
    - Use hash join
    - Hash the stream with smaller cardinality

There are basically 3 possible join algorithms that will be used for 3 cases:
Both streams are IndexRetrievals,
One stream is an IndexRetrieval,
And both streams are non-indexed arbitrary streams.

It is important to note that if there is an index as input it is always used.

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



38

See slide 29, the diagram is the same.

Joins use sideways information passing by taking information about indices where indices are present on streams as input.

This is not done if the input to a Join is a Join or another filter.

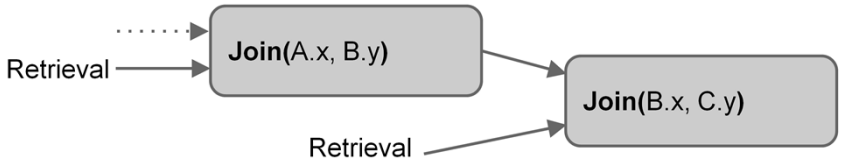
Multi-Predicate Join Optimization

```
// 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()),
        OdetailTable.joinOnFieldOno()),
    new JoinPredicate(
        OrderTable.joinOnFieldEno(),
        EmpTable.joinOnItself())));
```


This slide only serves to show what the API looks like for multi-predicate equijoins.

Multi-Predicate Join Algorithm

- Users can also specify their own join execution plans
  - Could have bushy joins
  - Or right-deep or left-deep joins
- By default, the algorithm joins performs the most optimal join at any step.



By default, the multi-predicate join algorithm finds the most appropriate equijoin predicate to process.

This can result in bushy joins, it is not enforced to be left-deep or right-deep.

The user can easily specify new join query plans.

In the future it would make sense for some of the joins to be done in parallel.

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
  - 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
  - Bytecode modification makes this simple!

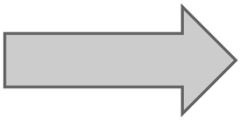
The first point means that the database could use reflection to retrieve private fields from other classes by playing around with the SecurityManager.

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**

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

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



```
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);}}
```


In order to really do due justice on this point, and eliminate a lot of the constraints on the source package,
we would need a framework which could detect updates to any of the fields of a class, and then inject
a statement to update the indexes related to MercuryDB

Table Field Declarations

- Every generated table class has a set whose elements are weakly referenced to store all instances of its mapped class
  - public static Set<WeakReference<Foo>> table;
- Since only id and name are indexed, two maps are generated
  - private static Map<Integer, Set<WeakReference<Foo>> idIndex;
  - private static Map<String, Set<WeakReference<Foo>>> nameIndex;

A WeakReference is Java magic: It's treated specially by the JVM so we don't have any direct control over its behavior,
but it gives us some nice properties for the sake of this project.

Note that for indexes the values (sets themselves) are weakly referenced as well. So that if an indexed instance is garbage collected, it goes away in the index as well as the table.

Database Consistency

- How are objects added to the database?
  - via bytecode modification
- How are indices kept consistent?
  - must use setters in database
- What happens when instances in the table go out of scope?
  - Garbage Collected (deleted)
  - Tables hold WeakReferences
  - Garbage Collector ignores the references in tables

By Database Consistency, we mean keeping the Mercury DB consistent with the client application as objects are created and updated.

For non-indexed objects, this is trivial because they are added to the Tables on creation, and those Tables reference these objects directly,
so we will always query on the most recent versions of these objects.

For indexed objects, must use setters as defined by the generated MercuryDB schema

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)

We wanted a way to map something we know well to the new system for testing, so we modified the MDB project to take MDB SQL queries and convert them into Target and Client modules that can be used to exercise the functionality of the API.

RetrogradeDB implements **IDatabase**

- Converts MDB SQL scripts to Java classes
- Modified MDB Project
- Same Parser
- New IDatabase implementation for Mercury Retrograde
- Mustache to format output

MDB script to Java code. These scripts will define both the target module and the client model of the resulting test code.

We reimplemented the MDB database interface (IDatabase) to turn the test scripts into Java code which we could use to test MercuryDB.

RetrogradeDB implements **IDatabase**

```
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();

}
```


- **create** and **index** result in the “target module” class definitions, which represent tables (I’ll show an example on the next slide)
- **insert**, **update**, **delete**, and **select** are all translated to operations in the client program.
- **commit** is processed to write the templates
- **abort** resets the data collected since the last write
- **close** has no meaning in this implementation because there is no active database to close.

RetrogradeDB - Target Module

- **create …;**
  - 1-to-1 target package class definition
- **index …;**
  - add @Index annotation to specified field in the generated class definition

RetrogradeDB - Target Mustache

```
public class {{relationName}} {
  {{#fields}}
  {{#isIndexed}}
  @Index
  {{/isIndexed}}
  {{#isInteger}}
  public int {{name}};
  {{/isInteger}}
  {{#isString}}
  public String {{name}};
  {{/isString}}
  {{/fields}}
```

```
  public {{relationName}}(
   {{#fields}}
   {{#isInteger}}
   int {{name}},
   {{/isInteger}}
   {{#isString}}
   String {{name}},
   {{/isString}}
   {{/fields}}
   )
  {
   {{#fields}}
   this.{{name}} = {{name}};
   {{/fields}}
  }
}
```


This is the template for generating the Target module class definitions [one for each relation] (these classes will be converted into tables in the generated MercuryDB fixed-schema API for this target package).

RetrogradeDB - Target Code

```
create table emp (
              empno int,
              age   int
          );
.
index emp.age;
.
commit;
.
```

```
public class Emp {
  public int empno;
  @Index
  public int age;

  public Emp(
    int empno,
    int age
  )
  {
    this.empno = empno;
    this.age = age;
  }
}
```


Here is an example of the a Target module class that is generated as a result of MDB SQL commands **create** and **index**

As you can see, the create statement is 1-to-1 with the generated class, and it has the same name
(the name was transformed to Java-standard camel-case capitalization [“emp” -> “Emp”])

The index statement is processed and an @Index annotation is added to the appropriate field in the output class.

RetrogradeDB - 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

The reason we keep the ArrayList of Objects is to keep active references so that they will remain in the database.
If the only references to the objects are in the database, they will be garbage collected.

RetrogradeDB - Client Module

- **select …;**
  - Return references to entire tuples
    - select [fields] = select *
    - simplifies code generation somewhat
  - Uses API generated by JavaDB

The select statement uses the API to retrieve the tuples qualifying for all the predicates in the WHERE clause,
and then displays them (which seems the logical thing to do for a select in SQL semantics).

Keep in mind that the code generated here is just an example that corresponds as closely as possible to the behavior of MDB,
However, the strategy we use to observe the results is only one of many possible ways to interact with the results.

RetrogradeDB - Client Code

```
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);  // set new value

i = EmpTable.queryEmpno(1);
for (Emp e : i.elements())  // select
    System.out.println(e);  // display

i = EmpTable.queryEmpno(2);
for (Emp e : i.elements())  // delete
    emps.remove(e);  // remove from app
```


Here is an example of the a Target module class that is generated as a result of MDB SQL commands **create** and **index**

We store the created objects in an ArrayList so that we can easily keep references to the objects we create (thus keeping them alive)
Of course, any data structure could be used, and Mercury will keep track of objects created and stored in any way
(including scalar variables e.g. Emp e = new Emp(1,25)).

The strategy for iterating over the returned results for the sake of select, update, and delete, is just one strategy which shows how to use the iterator to access all the returned records in order.

Outline

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

References

[1] Zachary G. Ives and Nicholas E. Taylor,
"Sideways Information Passing for Push-Style Query Processing," in CIS, 2008.
http://repository.upenn.edu/cgi/viewcontent.cgi?article=1045&context=db_research

[2] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price,
"Access Path Selection in a Relational Database System," in ACM, 1979
http://dl.acm.org/citation.cfm?doid=582095.582099

Questions?