Goals for Today

- Learn about how relational algebra operates on sets of tuples
- Compose the basic relational algebra operators to form *queries* on relations
- Understand the goals for Lab 2
  — You’re ready to start after class today!

*Logical Query Plan Example*

- Example: college database

  ```
  Students(SID, name, gpa)
  Enrolled(SID, CID, grade)
  ```

  ```
  SELECT S.name, E.CID
  FROM Students S, Enrolled E
  WHERE S.SID=E.SID;
  ```

  *Sets of tuples flow upward*

  - Get tuples from Students
  - Get tuples from Enrolled
  - Combine
  - Pull out name and CID fields

  When query optimizer forms *physical query plan*, it will consider available implementations and/or choices for the logical operators!

*Operations on Sets of Tuples*

- **Relational model**: data represented as sets of tuples (i.e., *relations*)
- **Relational algebra**: an *algebra* on sets of tuples
  - Used to express *queries* about those relations
  - I.e., a *query language*

- **Note: Sets != Bags**
  - Sets: relations have no duplicate tuples
  - Bags, aka multi-sets: duplicate tuples possible
Formal Relational Query Languages

- *Query languages* allow manipulation and retrieval of data from a database
  - Query languages $\neq$ programming languages!

- Two mathematical Query Languages form the basis for “real” languages (e.g., SQL), and for implementation:
  - *Relational Algebra*: More operational, very useful for representing query execution plans.
  - *Relational Calculus*: Lets users describe what they want, rather than how to compute it. (Non-operational, declarative.)

We’ll see some differences between SQL and relational algebra

What is “an Algebra” ??

Mathematical system consisting of:

<table>
<thead>
<tr>
<th>Variables or constants</th>
<th>Operands</th>
<th>Operators</th>
</tr>
</thead>
</table>

Example:

- Arithmetic: $x, y, 15$, $+, -, *, /$

- Relational Relations Let’s see...

An algebra allows us to build expressions by applying operators to operands and/or other expressions

Example Instances

**Sailing Database:**
Boats, Sailors, Reserves

<table>
<thead>
<tr>
<th>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

**Reserves**

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

**Boats**

<table>
<thead>
<tr>
<th>bid</th>
<th>bname</th>
<th>color</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>Interlake</td>
<td>blue</td>
</tr>
<tr>
<td>102</td>
<td>Interlake</td>
<td>red</td>
</tr>
<tr>
<td>103</td>
<td>Clipper</td>
<td>green</td>
</tr>
<tr>
<td>104</td>
<td>Marine</td>
<td>red</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>yuppy</td>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>44</td>
<td>guppy</td>
<td>5</td>
<td>35.0</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

Preliminaries

A query is applied to *relation instances*, and the result of a query is also a relation instance.

Depending on the query, output relation schema may be the same or different than input schema

<table>
<thead>
<tr>
<th>SID</th>
<th>name</th>
<th>login</th>
<th>gpa</th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
<td>Alice</td>
<td>alicious</td>
<td>3.4</td>
</tr>
<tr>
<td>67</td>
<td>Bob</td>
<td>bobtastic</td>
<td>3.9</td>
</tr>
<tr>
<td>78</td>
<td>Carl</td>
<td>carl</td>
<td>2.5</td>
</tr>
</tbody>
</table>

Query (e.g., algebra or SQL)
Relational Algebra: 5 Basic Operations

- **Selection** (\(\sigma\)) Selects a subset of *rows* from relation (horizontal).
- **Projection** (\(\pi\)) Retains only wanted *columns* from relation (vertical).
- **Cross-product** (\(\times\)) Allows us to combine two relations.
- **Set-difference** (\(-\)) Tuples in relation1, but not in relation2.
- **Union** (\(\cup\)) Tuples in relation1 and/or in relation2.

Since each operation returns a relation, operations can be composed!

### Selection (\(\sigma\)) – Horizontal Restriction

- Selects rows that satisfy *selection condition*.
  - Note: not the same thing as SELECT in SQL
  - Can have several conditions, combined with \(\lor\) (or), \(\land\) (and)

- **Schema** of result is same as that of the input relation.

- Example:

<table>
<thead>
<tr>
<th>sid</th>
<th>surname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>yuppy</td>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>44</td>
<td>guppy</td>
<td>5</td>
<td>35.0</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

\(\sigma_{rating=8}(S2)\)

### Projection (\(\pi\)) – Vertical Restriction

- **Examples:** \(\pi_{age}(S2)\) \(\pi_{surname,rating}(S2)\)

- Retains only attributes that are in the “projection list”.

- **Schema** of result:
  - exactly the fields in the projection list, with the same names that they had in the input relation

- In relational algebra, projection operator **eliminates duplicates**
  - How would duplicates arise?
  - Note: real systems typically don’t do duplicate elimination in SQL unless the user explicitly asks for it (why not?)

### Examples: Projection

<table>
<thead>
<tr>
<th>sid</th>
<th>surname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>yuppy</td>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>44</td>
<td>guppy</td>
<td>5</td>
<td>35.0</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

\(\pi_{age}(S2)\)

\(\pi_{surname,rating}(S2)\)
Composing Operators

- Output of a Relational Algebra operator is a relation, so...
  - Can use result as input to another Relational Algebra operator

\[
\pi_{\text{name, rating}}(\sigma_{\text{rating}>8}(S2))
\]

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>yuppy</td>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>44</td>
<td>guppy</td>
<td>5</td>
<td>35.0</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

Union and Set-Difference

- Take two input relations, which must be union-compatible:
  - Same number of fields
  - “Corresponding” fields have the same type

Will the Union operator have to do duplicate elimination? How about Set-Difference?

Union

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>yuppy</td>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>44</td>
<td>guppy</td>
<td>5</td>
<td>35.0</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

Set-difference

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

\[
S1 \cup S2
\]

\[
S1 - S2
\]

\[
S2 - S1
\]
Cross-Product

- S1 x R1: Each tuple of S1 paired with each tuple of R1

How many tuples will be in the result S1 x R1?

- Result schema has one field per field of S1 and R1, with field names `inherited` if possible.
  - May have a naming conflict: When both S1 and R1 have a field with the same name.
  - In this case, can use the renaming operator:
    \[ \rho(C(l \rightarrow sid1, 5 \rightarrow sid2), S1 \times R1) = \]
    \[
    \begin{array}{ccc}
    sid1 & sname & rating & age \\
    22 & dustin & 7 & 45.0 \\
    31 & lubber & 8 & 55.5 \\
    58 & rusty & 10 & 35.0 \\
    \end{array}
    \]

Compound Operators

- In addition to the five basic operators, there are several additional “Compound Operators”
  - These add no computational power to the language, but are useful short-hands
  - Can be expressed solely with the basic operators

- We’ll look at
  - Intersection, Join
- See book for
  - Division

Cross Product Example

<table>
<thead>
<tr>
<th>S1</th>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td></td>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td></td>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R1</th>
<th>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td></td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

\[ \rho(C(l \rightarrow sid1, 5 \rightarrow sid2), S1 \times R1) = \]

Intersection

Intersection takes two input relations, which must be union-compatible.

- How to express it using only basic operators?
  \[ R \cap S = R - (R - S) \]

<table>
<thead>
<tr>
<th>S1</th>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td></td>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td></td>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S1</th>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td></td>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

\[ S1 \cap S2 \]
Join (⋈)

- Joins are compound operators involving
  - cross product,
  - selection,
  - and (sometimes) projection.

- Most common type of join is a natural join
  (bowtie with no annotation)

  $R \bowtie S$ conceptually is:
  - Compute the cross product $R \times S$
  - Select rows where attributes that appear in both relations have equal values
  - Project all unique attributes and one copy of each of the common ones

Other Types of Joins

- Condition Join (or “theta-join”):
  $R \bowtie_C S = \sigma_C (R \times S)$
  - Result schema same as that of cross-product.
  - (May have fewer tuples than cross-product)

- Equi-Join: Nickname for case when condition $c$ contains only conjunction of equalities.
Exercise 5-7:
Relational Algebra using Joins

Find names of sailors who’ve reserved boat #103
\[ \pi_{\text{name}}((\sigma_{\text{bid}=103} \text{Reserves}) \bowtie \text{Sailors}) \]

Find names of sailors who’ve reserved a red boat
\[ \pi_{\text{name}}((\sigma_{\text{color}='\text{red}'} \text{Boats}) \bowtie \text{Reserves} \bowtie \text{Sailors}) \]

Find sailors who’ve reserved a red and a green boat
\[ \rho (\text{Tempred}, \pi_{\text{sid}}((\sigma_{\text{color}='\text{red}'} \text{Boats}) \bowtie \text{Reserves})) \]
\[ \rho (\text{Tempgreen}, \pi_{\text{sid}}((\sigma_{\text{color}='\text{green}'} \text{Boats}) \bowtie \text{Reserves})) \]
\[ \pi_{\text{name}}((\text{Tempred} \cap \text{Tempgreen}) \bowtie \text{Sailors}) \]

Lab 2: SimpleDb Operators

- **Goal**: building on Lab 1, be able to perform simple queries over multiple relations
  
  ```sql
  SELECT *
  FROM table1, table2
  WHERE table1.field1 = table2.field2
  AND table1.id > 5;
  ```

- Three parts! Submit each on Sakai
  - Part 1: Filter and Join: out now, due next Wednesday
  - Part 2: Aggregation, HeapFile Mutability
  - Final: Insert and Delete, Buffer Pool Eviction

- Operators will implement the interface `DbIterator`

---

Operators are DbIterators

```java
public abstract class Operator implements DbIterator {
    // hasNext() and next() are already written
    public boolean hasNext() throws DbException, TransactionAbortedException;
    public Tuple next() throws DbException, TransactionAbortedException;
    // NoSuchElementException
    if (next == null) {
        next = fetchNext();
        if (next == null) throw new NoSuchElementException();
    }
    Tuple result = next;
    next = null;
    return result;
}
protected abstract Tuple fetchNext();
```

Example: Filter

```java
public class Filter extends Operator {
    // Constructor accepts a predicate to apply
    // and a child operator to read tuples to filter from.
    Predicate p;
    Operator child;

    // The predicate to filter tuples with
    public Filter(Predicate p, DbIterator child) {
    // A child operator from which to read tuples!
    // You will need to call super.open() and super.close()
    // Calls fetchNext()
    // Your operators will have to implement this
    Hint: check out Field.compare()
    }
}
```

See Project.java as an example
Join

Takes a JoinPredicate:

```java
public JoinPredicate(int field1, Predicate.Op op, int field2) {
    Which field from the tuple from child1
    Which field from the tuple from child2
```

Recall nested-loop algorithm...

What will the TupleDesc look like for the tuples output by JOIN?