Relational Calculus

CS 186, Fall 2005
R&G, Chapter 4

We will occasionally use this arrow notation unless there is danger of no confusion.
Ronald Graham
Elements of Ramsey Theory

Relational Calculus

• Comes in two flavors: Tuple relational calculus (TRC) and Domain relational calculus (DRC).
• Calculus has variables, constants, comparison ops, logical connectives and quantifiers.
  - TRC: Variables range over (i.e., get bound to) tuples.
    - Like SQL.
  - DRC: Variables range over domain elements (= field values).
    - Like Query-By-Example (QBE)
  - Both TRC and DRC are simple subsets of first-order logic.
    • We’ll focus on TRC here
• Expressions in the calculus are called formulas.
• Answer tuple is an assignment of constants to variables that make the formula evaluate to true.

Tuple Relational Calculus

• **Query** has the form: \( \{ T | p(T) \} \)
  - \( p(T) \) denotes a formula in which tuple variable \( T \) appears.
• **Answer** is the set of all tuples \( T \) for which the formula \( p(T) \) evaluates to true.
• **Formula** is recursively defined:
  ✓ start with simple atomic formulas (get tuples from relations or make comparisons of values)
  ✓ build bigger and better formulas using the logical connectives.

TRC Formulas

• An Atomic formula is one of the following:
  - \( R \in Rel \)
  - \( R.a \ op S.b \)
  - \( R.a \ op \text{ constant} \)
    - \( op \) is one of \( <, >, =, \leq, \geq, \neq \)
• A formula can be:
  - an atomic formula
  - \( \neg p, p \land q, p \lor q \) where \( p \) and \( q \) are formulas
  - \( \exists R(p(R)) \) where variable \( R \) is a tuple variable
  - \( \forall R(p(R)) \) where variable \( R \) is a tuple variable

Free and Bound Variables

• The use of quantifiers \( \exists X \) and \( \forall X \) in a formula is said to bind \( X \) in the formula.
  - A variable that is not bound is free.
• Let us revisit the definition of a query:
  - \( \{ T | p(T) \} \)

• There is an important restriction
  - the variable \( T \) that appears to the left of \( '|' \) must be the only free variable in the formula \( p(T) \).
  - in other words, all other tuple variables must be bound using a quantifier.

Selection and Projection

• Find all sailors with rating above 7
  \( \{ S | S \in Sailors \land S.rating > 7 \} \)
  - Modify this query to answer: Find sailors who are older than 18 or have a rating under 9, and are called ‘Bob’.
• Find names and ages of sailors with rating above 7.
  \( \{ S | \exists S1 \in Sailors(S1.rating > 7 \land S.sname = S1.sname \land S.age = S1.age) \} \)
  - Note: \( S \) is a tuple variable of 2 fields (i.e. \( S \) is a projection of \( S\)ailors)
    • only 2 fields are ever mentioned and \( S \) is never used to range over any relations in the query.
Joins

Find sailors rated > 7 who’ve reserved boat #103

\{ S \in \text{Sailors} \land S.\text{rating} > 7 \land \\
\exists R(R \in \text{Reserves} \land R.sid = S.sid \\
\land R.bid = 103) \}

Note the use of \( \exists \) to find a tuple in Reserves that "joins with" the Sailors tuple under consideration.

Joins (continued)

\{ S \in \text{Sailors} \land S.\text{rating} > 7 \land \\
\exists R(R \in \text{Reserves} \land R.sid = S.sid \\
\land R.bid = 103) \land \\
\exists B(B \in \text{Boats} \land B.bid = R.bid \\
\land B.\text{color} = \text{'red'}) \}

Find sailors rated > 7 who’ve reserved a red boat

- Observe how the parentheses control the scope of each quantifier’s binding.
- This may look cumbersome, but it’s not so different from SQL!

Division (makes more sense here???)

Find sailors who’ve reserved all boats
(hint, use \( \forall \))

\{ S \in \text{Sailors} \land \\
\forall B(B \in \text{Boats} \land \exists R(R \in \text{Reserves} \\
\land S.sid = R.sid \\
\land B.bid = R.bid)) \}

- Find all sailors \( S \) such that for all tuples \( B \) in Boats there is a tuple in Reserves showing that sailor \( S \) has reserved \( B \).

Division – a trickier example...

Find sailors who’ve reserved all Red boats

\{ S \in \text{Sailors} \land \\
\forall B(B \in \text{Boats} \land B.\text{color} = \text{'red'} \Rightarrow \\
\exists R(R \in \text{Reserves} \land S.sid = R.sid \\
\land B.bid = R.bid)) \}

Alternatively...

\{ S \in \text{Sailors} \land \\
\forall B(B \in \text{Boats} \land B.\text{color} \ne \text{'red'} \land \\
\exists R(R \in \text{Reserves} \land S.sid = R.sid \\
\land B.bid = R.bid)) \}

Unsafe Queries, Expressive Power

- \( \exists \) syntactically correct calculus queries that have an infinite number of answers! Unsafe queries.
  - e.g., \( \med \neg \{ S \in \text{Sailors} \} \)
  - Solution??? Don’t do that!
- Expressive Power (Theorem due to Codd):
  - every query that can be expressed in relational algebra can be expressed as a safe query in DRC / TRC; the converse is also true.
- Relational Completeness: Query language (e.g., SQL) can express every query that is expressible in relational algebra/calculus. (actually, SQL is more powerful, as we will see...)

a \Rightarrow b \text{ is the same as } \neg a \lor b

<table>
<thead>
<tr>
<th></th>
<th>T</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

- If \( a \) is true, \( b \) must be true!
  - If \( a \) is true and \( b \) is false, the implication evaluates to false.
- If \( a \) is not true, we don’t care about \( b \)
  - The expression is always true.
**Summary**

- The relational model has rigorously defined query languages — simple and powerful.
- Relational algebra is more operational
  - useful as internal representation for query evaluation plans.
- Relational calculus is non-operational
  - users define queries in terms of what they want, not in terms of how to compute it. *(Declarative)*
- Several ways of expressing a given query
  - a query optimizer should choose the most efficient version.
- Algebra and safe calculus have same expressive power
  - leads to the notion of relational completeness.

**Addendum: Use of ∀**

- ∀x (P(x)) - is only true if P(x) is true for every x in the universe
- Usually:
  - ∀x ((x ∈ Boats) ⇒ (x.color = “Red”))
  - ⇒ logical implication,
    - a ⇒ b means that if a is true, b must be true
    - a ⇒ b is the same as ¬a ∨ b

**Find sailors who’ve reserved all boats**

\[
\{S | S ∈ Sailors \land \forall B((B ∈ Boats) \implies \exists R((R ∈ Reserves \land S.sid = R.sid \land B.bid = R.bid)))\}
\]

**Find sailors who’ve reserved all boats**

\[
\{S | S ∈ Sailors \land \forall B((B ∈ Boats) \lor \exists R((R ∈ Reserves \land S.sid = R.sid \land B.bid = R.bid)))\}
\]

**Relational Query Languages**

- A major strength of the relational model: supports simple, powerful querying of data.
- Two sublanguages:
  - DDL – Data Definition Language
    - define and modify schema (at all 3 levels)
  - DML – Data Manipulation Language
    - Queries can be written intuitively.
- The DBMS is responsible for efficient evaluation.
  - The key: precise semantics for relational queries.
  - Allows the optimizer to re-order/change queries, and ensure that the answer does not change.
  - Internal cost model drives use of indexes and choice of access paths and physical operators.

---

**SQL: The Query Language Part 1**

CS186, Fall 2005
R&G, Chapter 5

Life is just a bowl of queries.

- Anon
  (not Forrest Gump)
The SQL Query Language

- The most widely used relational query language.
  - Current standard is SQL-1999
  - Not fully supported yet
  - Introduced "Object-Relational" concepts (and lots more)
    - Many of which were pioneered in Postgres here at Berkeley!
  - SQL-200x is in draft
  - SQL-92 is a basic subset
  - Most systems support a medium
  - PostgreSQL has some "unique" aspects
    - as do most systems.
  - XML support/integration is the next challenge for SQL
    (more on this in a later class).

Create Table (w/column constraints)

- CREATE TABLE table_name
  { [ column_name ] data_type [ DEFAULT default_expr ] [ column_constraint [ , ... ] ] [ table_constraint [ , ... ] ] }

  Column Constraints:
  - [ CONSTRAINT constraint_name ]
    { NOT NULL | NULL | UNIQUE | PRIMARY KEY | CHECK (expression) }
    REFERENCES ref_table [ ( refcolumn ) ] [ ON DELETE action ] [ ON UPDATE action ]
  - action is one of:
  - NO ACTION, CASCADE, SET NULL, SET DEFAULT
  - expression for column constraint must produce a boolean result and reference the related column's value only.

Create Table (w/table constraints)

- CREATE TABLE table_name
  { [ column_name ] data_type [ DEFAULT default_expr ] [ column_constraint [ , ... ] ] [ table_constraint [ , ... ] ] }

  Table Constraints:
  - [ CONSTRAINT constraint_name ]
    { UNIQUE ( column_name [ , ... ] ) | PRIMARY KEY ( column_name [ , ... ] ) | CHECK (expression) | FOREIGN KEY ( column_name [ , ... ] ) REFERENCES ref_table [ ( refcolumn [ , ... ] ) ] [ ON DELETE action ] [ ON UPDATE action ] }

  Here, expressions, keys, etc can include multiple columns

Create Table (Examples)

CREATE TABLE films (  
code CHAR(5) PRIMARY KEY,  
title VARCHAR(40),  
did DECIMAL(3),  
date_prod DATE,  
kind VARCHAR(10),  
CONSTRAINT production UNIQUE(date_prod)  
FOREIGN KEY did REFERENCES distributors  
ON DELETE NO ACTION );

CREATE TABLE distributors (  
did DECIMAL(3) PRIMARY KEY,  
name VARCHAR(40)  
CONSTRAINT con1 CHECK (did > 100 AND name <> '' ) );

The SQL DML

- Single-table queries are straightforward.
- To find all 18 year old students, we can write:

```
SELECT * FROM Students S  
WHERE S.age=18
```

- To find just names and logins, replace the first line:
  SELECT S.name, S.login

• Data Types (PostgreSQL) include:
  - character(n) – fixed-length character string
  - character varying(n) – variable-length character string
  - smallint, integer, bigint, numeric, real, double precision
  - date, time, timestamp, ...
  - serial: unique ID for indexing and cross reference

• PostgreSQL also allows OIDs, arrays, inheritance, rules…
  - conformance to the SQL-1999 standard is variable so we won’t use these in the project.
Querying Multiple Relations

- Can specify a join over two tables as follows:

```
SELECT S.name, E.cid
FROM Students S, Enrolled E
WHERE S.sid=E.sid AND E.grade='B'
```

<table>
<thead>
<tr>
<th>sid</th>
<th>cid</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>Carnatic101</td>
<td>C</td>
</tr>
<tr>
<td>53831</td>
<td>Reggae203</td>
<td>B</td>
</tr>
<tr>
<td>53650</td>
<td>Topology112</td>
<td>A</td>
</tr>
<tr>
<td>53666</td>
<td>History105</td>
<td>B</td>
</tr>
</tbody>
</table>

Note: obviously no referential integrity constraints have been used here.

Basic SQL Query

- `relation-list`: A list of relation names
  - possibly with a `range-variable` after each name
- `target-list`: A list of attributes in `relation-list`
- `qualification`: Comparisons combined using AND, OR and NOT.
  - Comparisons are Attr1 `op` Attr2 or Attr1 `op` Attr2, where `op` is one of `<`, `>`, `=`, `<=`, `>=`.
- `DISTINCT`: optional keyword indicating that the answer should not contain duplicates.
  - In SQL SELECT, the default is that duplicates are NOT eliminated! (Result is called a “multiset”)

Query Semantics

- Semantics of an SQL query are defined in terms of the following conceptual evaluation strategy:
  1. do `FROM` clause: compute cross-product of tables (e.g., Students and Enrolled).
  2. do `WHERE` clause: Check conditions, discard tuples that fail. (called “selection”).
  3. do `SELECT` clause: Delete unwanted fields. (called “projection”).
  4. If `DISTINCT` specified, eliminate duplicate rows.
- Probably the least efficient way to compute a query!
  - An optimizer will find more efficient strategies to get the same answer.

Step 1 – Cross Product

```
SELECT S.name, E.cid
FROM Students S, Enrolled E
WHERE S.sid=E.sid AND E.grade='B'
```

Step 2) Discard tuples that fail predicate

```
SELECT S.name, E.cid
FROM Students S, Enrolled E
WHERE S.sid=E.sid AND E.grade='B'
```

Step 3) Discard Unwanted Columns

```
SELECT S.name, E.cid
FROM Students S, Enrolled E
WHERE S.sid=E.sid AND E.grade='B'
```
Now the Details

We will use these instances of relations in our examples.

(Question: If the key for the Reserves relation contained only the attributes sid and bid, how would the semantics differ?)

<table>
<thead>
<tr>
<th>Reserves</th>
<th>Sailors</th>
<th>Boats</th>
</tr>
</thead>
<tbody>
<tr>
<td>sid</td>
<td>bid</td>
<td>day</td>
</tr>
<tr>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>95</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sailors</th>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>Dustin</td>
<td>7</td>
<td>45.0</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>Lubber</td>
<td>8</td>
<td>55.5</td>
<td></td>
</tr>
<tr>
<td>95</td>
<td>Bob</td>
<td>3</td>
<td>63.5</td>
<td></td>
</tr>
</tbody>
</table>

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

SELECT sname
FROM Sailors, Reserves
WHERE Sailors.sid=Reserves.sid
AND bid=103

Example Schemas

CREATE TABLE Sailors (sid INTEGER PRIMARY KEY, sname CHAR(20), rating INTEGER, age REAL)

CREATE TABLE Boats (bid INTEGER PRIMARY KEY, bname CHAR (20), color CHAR(10))

CREATE TABLE Reserves (
  sid INTEGER REFERENCES Sailors, bid INTEGER, day DATE, PRIMARY KEY (sid, bid, day), FOREIGN KEY (bid) REFERENCES Boats)

Another Join Query

SELECT sname
FROM Sailors, Reserves
WHERE Sailors.sid=Reserves.sid
AND bid=103

<table>
<thead>
<tr>
<th>(sid) sname</th>
<th>rating</th>
<th>age</th>
<th>(sid) bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22 dustin</td>
<td>7</td>
<td>45.0</td>
<td>22</td>
<td>101</td>
</tr>
<tr>
<td>22 dustin</td>
<td>7</td>
<td>45.0</td>
<td>95</td>
<td>103</td>
</tr>
<tr>
<td>31 lubber</td>
<td>8</td>
<td>55.5</td>
<td>22</td>
<td>101</td>
</tr>
<tr>
<td>31 lubber</td>
<td>8</td>
<td>55.5</td>
<td>95</td>
<td>103</td>
</tr>
<tr>
<td>95 Bob</td>
<td>3</td>
<td>63.5</td>
<td>22</td>
<td>101</td>
</tr>
<tr>
<td>95 Bob</td>
<td>3</td>
<td>63.5</td>
<td>95</td>
<td>103</td>
</tr>
</tbody>
</table>

Some Notes on Range Variables

- Can associate "range variables" with the tables in the FROM clause.
  - saves writing, makes queries easier to understand
- Needed when ambiguity could arise.
  - for example, if same table used multiple times in same FROM (called a "self-join")

SELECT sname
FROM Sailors, Reserves
WHERE Sailors.sid=Reserves.sid AND bid=103

Can be rewritten using range variables as:

SELECT S.sname
FROM Sailors S, Reserves R
WHERE S.sid=R.sid AND bid=103

More Notes

- Here's an example where range variables are required (self-join example):

  SELECT x.sname, x.age, y.sname, y.age
  FROM Sailors x, Sailors y
  WHERE x.age > y.age

- Note that target list can be replaced by "*" if you don't want to do a projection:

  SELECT *
  FROM Sailors x
  WHERE x.age > 20

Find sailors who've reserved at least one boat

SELECT S.sid
FROM Sailors S, Reserves R
WHERE S.sid=R.sid

- Would adding DISTINCT to this query make a difference?
- What is the effect of replacing S.sid by S.sname in the SELECT clause?
  - Would adding DISTINCT to this variant of the query make a difference?
### Expressions
- Can use arithmetic expressions in SELECT clause (plus other operations we’ll discuss later)
- Use AS to provide column names

```
SELECT S.age, S.age-5 AS age1, 2*S.age AS age2
FROM Sailors S
WHERE S.name = 'Dustin'
```

- Can also have expressions in WHERE clause:

```
SELECT S1.sname AS name1, S2.sname AS name2
FROM Sailors S1, Sailors S2
WHERE 2*S1.rating = S2.rating - 1
```

### String operations
- SQL also supports some string operations
- "LIKE" is used for string matching.

```
SELECT S.age, S.age-5 AS age1, 2*S.age AS age2
FROM Sailors S
WHERE S.sname LIKE 'B_3b'
```

- ‘_’ stands for any one character and ‘%’ stands for 0 or more arbitrary characters.

### Find sid’s of sailors who’ve reserved a red or a green boat
- UNION: Can be used to compute the union of any two union-compatible sets of tuples (which are themselves the result of SQL queries).

#### Vs.

SELECT R.sid
FROM Boats B, Reserves R
WHERE R.bid=B.bid AND
(B.color='red' OR B.color='green')

SELECT R.sid
FROM Boats B, Reserves R
WHERE R.bid=B.bid AND B.color='red'
UNION
SELECT R.sid
FROM Boats B, Reserves R
WHERE R.bid=B.bid AND B.color='green'

### AND Continued...
- INTERSECT: discussed in book. Can be used to compute the intersection of any two union-compatible sets of tuples.
- Also in text: EXCEPT (sometimes called MINUS)
- Included in the SQL/92 standard, but many systems don’t support them.
  - But PostgreSQL does!

```
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
WHERE S.sid=R.sid
AND R.bid=B.bid
AND B.color='red'
INTERSECT
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
WHERE S.sid=R.sid
AND R.bid=B.bid
AND B.color='green'
```

### Nested Queries
- Powerful feature of SQL: WHERE clause can itself contain an SQL query!
  - Actually, so can FROM and HAVING clauses.

**Names of sailors who’ve reserved boat #103:**

```
SELECT S.sname
FROM Sailors S
WHERE S.sid IN (SELECT R.sid
FROM Reserves R
WHERE R.bid=103)
```

- To find sailors who’ve not reserved #103, use NOT IN.
- To understand semantics of nested queries:
  - think of a nested loops evaluation: For each Sailors tuple, check the qualification by computing the subquery.
Nested Queries with Correlation

Find names of sailors who’ve reserved boat #103:

```sql
SELECT S.sname
FROM Sailors S
WHERE EXISTS (SELECT * FROM Reserves R
WHERE R.bid=103 AND S.sid=R.sid)
```

- ** EXISTS** is another set comparison operator, like **IN**.
- Can also specify **NOT EXISTS**
- If **UNIQUE** is used, and * is replaced by **R.bid**, finds sailors with at most one reservation for boat #103.
  - unique checks for duplicate tuples in a subquery;
- Subquery must be recomputed for each Sailors tuple.
  - Think of subquery as a function call that runs a query!

More on Set-Comparison Operators

- We’ve already seen **IN**, **EXISTS** and **UNIQUE**. Can also use **NOT IN**, **NOT EXISTS** and **NOT UNIQUE**.
- Also available: **op ANY, op ALL**
- Find sailors whose rating is greater than that of some sailor called Horatio:

```sql
SELECT *
FROM Sailors S
WHERE S.rating > ANY (SELECT s2.rating
FROM Sailors s2
WHERE s2.sname='Horatio')
```

Rewriting INTERSECT Queries Using IN

Find sid’s of sailors who’ve reserved both a red and a green boat:

```sql
SELECT R.sid
FROM Boats B, Reserves R
WHERE B.bid=R.bid
AND B.color='red'
AND R.sid IN (SELECT R2.sid
FROM Boats B2, Reserves R2
WHERE R2.bid=B2.bid
AND B2.color='green')
```

- Similarly, **EXCEPT** queries re-written using **NOT IN**.
- How would you change this to find names (not sid’s) of Sailors who’ve reserved both red and green boats?

Division in SQL

Find sailors who’ve reserved all boats:

```sql
SELECT S.sname
FROM Sailors S, Sailors S2
WHERE NOT EXISTS (SELECT B.bid
FROM Boats B
WHERE NOT EXISTS (SELECT R.bid
FROM Reserves R
WHERE R.bid=B.bid
AND R.sid=S.sid))
```

Basic SQL Queries - Summary

- An advantage of the relational model is its well-defined query semantics.
- SQL provides functionality close to that of the basic relational model.
  - some differences in duplicate handling, null values, set operators, etc.
- Typically, many ways to write a query
  - the system is responsible for figuring a fast way to actually execute a query regardless of how it is written.
- Lots more functionality beyond these basic features. Will be covered in subsequent lectures.

Aggregate Operators

- **Significant extension of relational algebra.**

```sql
SELECT COUNT(*)
FROM Sailors S
```

```sql
SELECT AVG(S.age)
FROM Sailors S
WHERE S.rating=10
```

```sql
SELECT COUNT(DISTINCT S.rating)
FROM Sailors S
WHERE S.sname='Bob'
```
### Aggregate Operators

| COUNT (*) |
| SUM (DISTINCT A) |
| AVG (DISTINCT A) |
| MAX (A) |
| MIN (A) |

\[ \text{single column} \]

#### SELECT example

- **First Query:**
  
  \[
  \text{SELECT } S.\text{name} \\
  \text{FROM Sailors } S \\
  \text{WHERE } S.\text{rating} = 10
  \]

- **Second Query:**
  
  \[
  \text{SELECT AVG (DISTINCT } S.\text{age}) \\
  \text{FROM Sailors } S \\
  \text{WHERE } S.\text{rating} = 10
  \]

### GROUP BY and HAVING

- **So far, we've applied aggregate operators to all (qualifying) tuples.**
  - Sometimes, we want to apply them to each of several groups of tuples.

- **Consider:** Find the age of the youngest sailor for each rating level.
  - In general, we don't know how many rating levels exist, and what the rating values for these levels are!
  - Suppose we know that rating values go from 1 to 10; we can write 10 queries that look like this (!):

    - For \( i = 1, 2, \ldots, 10 \):
      
      \[
      \text{SELECT MIN (S.age) \\
      FROM Sailors } S \\
      \text{WHERE } S.\text{rating} = i
      \]

### Find name and age of the oldest sailor(s)

- **The first query is incorrect!**
  
  \[
  \text{SELECT } S.\text{name}, \text{MAX (S.age)} \\
  \text{FROM Sailors } S
  \]

- **Third query equivalent to second query**
  - allowed in SQL/92 standard, but not supported in some systems.
  - \* PostgreSQL seems to run it

  \[
  \text{SELECT } S.\text{name}, S.\text{age} \\
  \text{FROM Sailors } S \\
  \text{WHERE } S.\text{age} = (\text{SELECT MAX (S2.\text{age}) \\
  \text{FROM Sailors } S2})
  \]

  \[
  \text{SELECT } S.\text{name}, S.\text{age} \\
  \text{FROM Sailors } S \\
  \text{WHERE } \left( \text{SELECT MAX (S2.\text{age})} \\
  \text{FROM Sailors } S2 \right) = S.\text{age}
  \]