# Relational Algebra

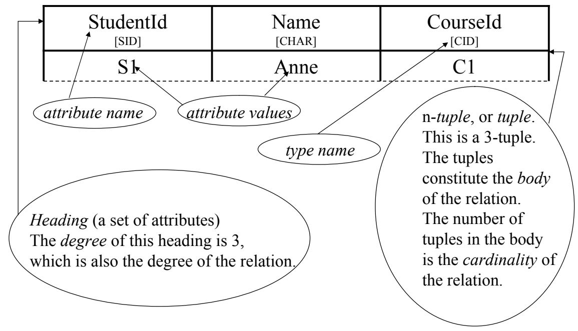
Hugh Darwen
(invited lecturer)

hughdarwen@gmail.com
www.dcs.warwick.ac.uk/~hugh

CS319: Relational Algebra
(revisited, reviewed, revised, simplified)

1

# Anatomy of a Relation




2

# Running Examples

IS_CALLED

| StudentId [SID] | Name [CHAR] |
|---|---|
| S1 | Anne |
| S2 | Boris |
| S3 | Cindy |
| S4 | Devinder |
| S5 | Boris |

Student *StudentId* is called *Name*

IS_ENROLLED_ON

| StudentId [SID] | CourseId [CID] |
|---|---|
| S1 | C1 |
| S1 | C2 |
| S2 | C1 |
| S3 | C3 |
| S4 | C1 |

Student *StudentId* is enrolled on course *CourseId*

3

# Relations and Predicates (1)

Consider the predicate: *StudentId* is called *Name*

… is called --- is the *intension* (meaning) of the predicate.

The parameter names are arbitrary. "*S* is called *N*" means the same thing (has the same intension).

The *extension* of the predicate is the set of *true* propositions that are *instantiations* of it:
{ S1 is called Anne, S2 is called Boris, S3 is called Cindy, S4 is called Devinder, S5 is called Boris }

Each tuple in the body of the relation provides the values to substitute for the parameters in one such instantiation.

4

# Relations and Predicates (2)

Moreover, each proposition in the extension has exactly one corresponding tuple in the relation.

This 1:1 correspondence reflects the *Closed World Assumption*:

> A tuple representing a true instantiation is in the relation.
> A tuple representing a false one is out.

The Closed World Assumption underpins the operators we are about to meet.

5

# Relational Algebra

Operators that operate on relations and return relations.

In other words, operators that are *closed over* relations. Just as arithmetic operators are closed over numbers.

Closure means that every invocation can be an operand, allowing expressions of arbitrary complexity to be written. Just as, in arithmetic, e.g., the invocation b-c is an operand of a+(b-c).

The operators of the relational algebra are relational counterparts of *logical* operators: AND, OR, NOT, EXISTS. Each, when invoked, yields a relation, which can be interpreted as the extension of some predicate.

6

## Logical Operators

Because relations are used to represent predicates, it makes sense for relational operators to be counterparts of operators on predicates. We will meet examples such as these:

Student *StudentId* is called *Name* **AND** *StudentId* is enrolled on course *CourseId*.

Student *StudentId* is enrolled **on some course**.

Student *StudentId* is enrolled on course *CourseId* AND *StudentId* is **NOT** called Devinder.

Student *StudentId* is NOT enrolled on any course **OR** *StudentId* is called Boris.

7

## Relational Operators

| Logic | Relational counterpart |
|---|---|
| AND | JOIN ($\bowtie$, *)<br>restriction (WHERE, $\sigma$)<br>extension(EXTEND)<br>summarization (SUMMARIZE) |
| EXISTS | projection (*r*{*attribute names*}, $\Pi$) |
| OR | UNION ($\cup$) |
| AND NOT | (semi)difference (NOT MATCHING, –) |
| | attribute renaming (RENAME, $\rho$) |

8

## A Bit of History

**1970**, E.F. Codd: Codd's algebra was incomplete (no extension, no attribute renaming) and somewhat flawed (Cartesian product).

**1975**, Hall, Hitchcock, Todd: *An Algebra of Relations for Machine Computation*. Fixed the problems, but not everybody noticed! Used in language ISBL.

**1998**, Date and Darwen: **Tutorial D**, a complete programming language, implemented in *Rel* (D. Voorhis). Relational operators based largely on ISBL.

**2011**, Elmasri and Navathe: *Database Systems*. Repeats Codd's flaw, offers flawed version of RENAME.

9

## JOIN (= AND)

*StudentId* is called *Name* **AND** *StudentId* is enrolled on *CourseId*.

IS_CALLED **JOIN** IS_ENROLLED_ON

IS_CALLED

| Name [CHAR] | StudentId [SID] |
|---|---|
| Anne | S1 |
| Boris | S2 |
| Cindy | S3 |
| Devinder | S4 |
| Boris | S5 |

IS_ENROLLED_ON

| StudentId [SID] | CourseId [CID] |
|---|---|
| S1 | C1 |
| S1 | C2 |
| S2 | C1 |
| S3 | C3 |
| S4 | C1 |

10

## IS_CALLED JOIN IS_ENROLLED_ON

| StudentId [SID] | Name [CHAR] | CourseId [CID] |
|---|---|---|
| S1 | Anne | C1 |
| S1 | Anne | C2 |
| S2 | Boris | C1 |
| S3 | Cindy | C3 |
| S4 | Devinder | C1 |

Note how this has "lost" the second Boris, not enrolled on any course.

11

## Definition of JOIN

Let $s = r1$ **JOIN** $r2$. Then:

The heading *Hs* of *s* is the union of the headings of *r1* and *r2*.

The body of *s* consists of those tuples having heading *Hs* that can be formed by taking the union of *t1* and *t2,* where *t1* is a tuple of *r1* and *t2* is a tuple of *r2*.

If *c* is a common attribute, then it must have the same declared type in both *r1* and *r2*. (I.e., if it doesn't, then *r1* JOIN *r2* is undefined.)

Note: JOIN, like AND, is both commutative and associative.

12

## RENAME

*Sid1* is called *Name*

IS_CALLED **RENAME** ( StudentId AS Sid1 )

| StudentId [SID] | Name [CHAR] |
|---|---|
| S1 | Anne |
| S2 | Boris |
| S3 | Cindy |
| S4 | Devinder |
| S5 | Boris |

| Sid1 [SID] | Name [CHAR] |
|---|---|
| S1 | Anne |
| S2 | Boris |
| S3 | Cindy |
| S4 | Devinder |
| S5 | Boris |

13

## Definition of RENAME

Let $s = r$ **RENAME** ( *A1* AS *B1*, ... *An* AS *Bn* )

The heading of *s* is the heading of *r* except that attribute *A1* is renamed to *B1* and so on.

The body of *s* consists of the tuples of *r* except that in each tuple attribute *A1* is renamed to *B1* and so on.

This definition stands in contrast to that offered by, e.g., Elmasri and Navathe. See the notes on this slide. Wikipedia gives a good definition, using $\rho$ as the operator name.

14

## RENAME and JOIN

*Sid1* is called *Name* AND so is *Sid2*

IS_CALLED RENAME (StudentId AS Sid1 ) JOIN
IS_CALLED RENAME (StudentId AS Sid2 )

| Sid1 [SID] | Name [CHAR] | Sid2 [SID] |
|---|---|---|
| S1 | Anne | S1 |
| S2 | Boris | S2 |
| S2 | Boris | S5 |
| S5 | Boris | S2 |
| S3 | Cindy | S3 |
| S4 | Devinder | S4 |
| S5 | Boris | S5 |

15

## Special Cases of JOIN

What is the result of *r* JOIN *r*?

*r*

What if all attributes are common to both operands?

It is called "intersection".

What if no attributes are common to both operands?

It is called "Cartesian product" (TIMES, X)

16

## Interesting Properties of JOIN

It is *commutative*: *r1* JOIN *r2* ≡ *r2* JOIN *r1*

It is *associative*: (*r1* JOIN *r2*) JOIN *r3* ≡ *r1* JOIN (*r2* JOIN *r3*)
So **Tutorial D** allows JOIN{*r1*, *r2*, …} (note the braces)

We note in passing that these properties are important for *optimisation* (in particular, of query evaluation).

Of course it is no coincidence that logical AND is also both commutative and associative.

17

## Projection (= EXISTS)

Student *StudentId* is enrolled **on some course**.

IS_ENROLLED_ON { StudentId }
= IS_ENROLLED_ON { ALL BUT CourseId }

Given:

| StudentId [SID] | CourseId [CID] |
|---|---|
| S1 | C1 |
| S1 | C2 |
| S2 | C1 |
| S3 | C3 |
| S4 | C1 |

To obtain:

| StudentId [SID] |
|---|
| S1 |
| S2 |
| S3 |
| S4 |

18

## Definition of Projection

Let *s* = *r* { *A1, ... An* }
( = *r* { ALL BUT *B1, ... Bm* } )

The heading of *s* is the subset of the heading of *r,* given by { *A1, ... An* }, equivalently by eliminating { *B1, ... Bm* }.

The body of *s* consists of each tuple that can be formed from a tuple of *r* by removing from it the attributes named *B1, ... Bm.*

Note that the cardinality of *s* can be less than that of *r* but cannot be more than that of *r.*

19

## Special Cases of Projection

What is the result of *r* { ALL BUT }?

*r*

What is the result of *r*{ }?

A relation with no attributes at all, of course!

There are two such relations, of cardinality 1 and 0. The pet names TABLE_DEE and TABLE_DUM have been advanced for these two, respectively.

20

## Special Case of AND (1)

*StudentId* is called *Name* **AND** *Name* begins with the letter *Initial*.

Given:

| StudentId [SID] | Name [CHAR] |
|---|---|
| S1 | Anne |
| S2 | Boris |
| S3 | Cindy |
| S4 | Devinder |
| S5 | Boris |

To obtain:

| StudentId [SID] | Name [CHAR] | Initial [CHAR] |
|---|---|---|
| S1 | Anne | A |
| S2 | Boris | B |
| S3 | Cindy | C |
| S4 | Devinder | D |
| S5 | Boris | B |

Much too difficult with JOIN. Why?

21

## Extension

*StudentId* is called *Name* **AND** *Name* begins with the letter *Initial*.

**EXTEND** IS_CALLED **: {** Initial := SUBSTRING (Name, 0, 1) }

Result:

| StudentId [SID] | Name [CHAR] | Initial [CHAR] |
|---|---|---|
| S1 | Anne | A |
| S2 | Boris | B |
| S3 | Cindy | C |
| S4 | Devinder | D |
| S5 | Boris | B |

22

## Definition of Extension

Let *s* = **EXTEND** *r* **:** { A1 := *exp1*, …, *An* := *expn* }

*exp1, ..., expn* are *open expressions,* mentioning attributes of *r*. The heading of *s* consists of the attributes of the heading of *r* plus the attributes *A1 ... An*. The declared type of attribute *Ak* is that of *exp-k.*

The body of *s* consists of tuples formed from each tuple of *r* by **add**ing *n* additional attributes *A1* to *An*. The value of attribute *Ak* is the result of evaluating *formula-k* on the corresponding tuple of *r*.

If we accept extension as primitive (which we must), then the formerly defined RENAME doesn't have to be regard as primitive. See the notes.

23

## Special Case of AND (2)

*StudentId* is called Boris

Can be done using JOIN and projection, like this:

( IS_CALLED JOIN
RELATION { TUPLE { Name NAME ( 'Boris' ) } } )
{ StudentId }

but it's easier using *restriction* (and projection again):

( IS_CALLED **WHERE** Name = NAME ('Boris' ) ) { StudentId }

result:

| StudentId |
|---|
| S2 |
| S5 |

"EXISTS *Name* such that *StudentId* is called *Name* AND *Name* is Boris"

24

## Definition of Restriction

Let *s* = *r* **WHERE** *c,* where *c* is a conditional expression on attributes of *r.*

The heading of *s* is the heading of *r*.

The body of *s* consists of those tuples of *r* for which the condition *c* evaluates to TRUE.

So the body of *s* is a subset of that of *r*.

Can also be defined in terms of previously defined operators (see the notes for this slide).

25

## Two More Relvars

COURSE

| CourseId [CID] | Title [CHAR] |
|---|---|
| C1 | Database |
| C2 | HCI |
| C3 | Op Systems |
| C4 | Programming |

*CourseId* is titled *Title*

EXAM_MARK

| StudentId [SID] | CourseId [CID] | Mark [INTEGER] |
|---|---|---|
| S1 | C1 | 85 |
| S1 | C2 | 49 |
| S2 | C1 | 49 |
| S3 | C3 | 66 |
| S4 | C1 | 93 |

*StudentId* scored *Mark* in the exam for course *CourseId*

26

## Aggregate Operators

An aggregate operator is one defined to operate on a relation and return a value obtained by aggregation over all the tuples of the operand. For example, simply to count the tuples:

COUNT ( IS_ENROLLED_ON ) = 5
COUNT ( IS_ENROLLED_ON
WHERE CourseId = CID ( 'C1' ) ) = 3

COUNT is an aggregate operator.

27

## More Aggregate Operators

SUM ( EXAM_MARK, Mark ) = 342

AVG ( EXAM_MARK, Mark ) = 68.4

MAX ( EXAM_MARK, Mark ) = 93

MIN ( EXAM_MARK, Mark ) = 49

MAX ( EXAM_MARK
WHERE CourseId = CID ( 'C2' ), Mark ) = 49

28

## Relations within a Relation

| CourseId [CID] | Exam_Result [RELATION{StudentID SID, Mark INTEGER}] |
|---|---|
| C1 | StudentId / Mark: S1 85; S2 49; S4 93 |
| C2 | StudentId / Mark: S1 49 |
| C3 | StudentId / Mark: S3 66 |
| C4 | StudentId / Mark |

Call this C_ER for future reference.

The Exam_Result values are called *image relations*, in EXAM_MARK, of tuples in COURSE.

29

## To obtain C_ER from COURSE and EXAM_MARK:

```
EXTEND COURSE ADD (
        ( EXAM_MARK JOIN
          RELATION { TUPLE { CourseId CourseId } } )
          { ALL BUT CourseId }
        AS Exam_Result )
{ CourseId, Exam_Result }
```

30

## Nested Relations and Agg Ops

The top score in the exam on course *CourseId* was *TopScore*

| CourseId [CID] | TopScore [INTEGER] |
|---|---|
| C1 | 93 |
| C2 | 49 |
| C3 | 66 |

EXTEND C_ER WHERE COUNT ( Exam_Result ) > 0 :
{TopScore := MAX ( Exam_Result, Mark )}
{ CourseId, TopScore }

Note the application of agg ops on image relations.

31

## SUMMARIZE BY

A shorthand for aggregation over image relations. For example, those top scores in each exam can be obtained directly from EXAM_MARK by:

**SUMMARIZE** EXAM_MARK **BY** { CourseId } :
{ TopScore := MAX ( Mark ) }

The usual first operand of the "agg op" is now omitted because it is implied by the combination of the SUMMARIZE operand (EXAM_MARK) and the BY operand ({CourseId }).

32

## SUMMARIZE PER

*Takers* is how many people took the exam on course *CourseId*

**SUMMARIZE** EXAM_MARK **PER** COURSE { CourseId } :
{ Takers := COUNT() }

result:

| CourseId [CID] | Takers [INTEGER] |
|---|---|
| C1 | 3 |
| C2 | 1 |
| C3 | 1 |
| C4 | 0 |

So EXAM_MARK **BY** { CourseId } is shorthand for EXAM_MARK **PER** EXAM_MARK { CourseId }.

33

## OR

*StudentId* is called *Name* **OR** *StudentId* is enrolled on *CourseId*.

| StudentId | Name | CourseId |
|---|---|---|
| S1 | Anne | C1 |
| S1 | Boris | C1 |
| S1 | Zorba | C1 |
| S1 | Anne | C4 |
| S1 | Anne | C943 |

and so on *ad infinitum* (almost!)

NOT SUPPORTED!

34

## UNION (restricted OR)

*StudentId* is called Devinder **OR** *StudentId* is enrolled on C1.

| StudentId |
|---|
| S1 |
| S2 |
| S4 |

(IS_CALLED WHERE Name = NAME ('Devinder')) { StudentId }
**UNION**
**(**IS_ENROLLED_ON WHERE CourseId = CID ('C1')) { StudentId }

35

## Definition of UNION

Let *s* = *r1* **UNION** *r2*. Then:

*r1* and *r2* must have the same heading.

The heading of *s* is the common heading of *r1* and *r2*.

The body of *s* consists of each tuple that is *either* a tuple of *r1* *or* a tuple of *r2*.

Is UNION commutative? Is it associative?

36

## NOT

*StudentId* is **NOT** called *Name*

| StudentId | Name |
|---|---|
| S1 | Boris |
| S1 | Quentin |
| S1 | Zorba |
| S1 | Cindy |
| S1 | Hugh |

and so on *ad infinitum* (almost!)

NOT SUPPORTED!

37

## Restricted NOT

*StudentId* is called *Name* **AND** is **NOT** enrolled on any course.

| StudentId | Name |
|---|---|
| S5 | Boris |

IS_CALLED **NOT MATCHING** IS_ENROLLED_ON

Sometimes referred to as "semidifference"

38

## Definition of NOT MATCHING

Let *s* = *r1* **NOT MATCHING** *r2*. Then:

The heading of *s* is the heading of *r1*.

The body of *s* consists of each tuple of *r1* that matches no tuple of *r2* on their common attributes.

It follows that in the case where there are no common attributes, *s* is equal to *r1* if *r2* is empty, and otherwise is empty.

When all attributes are common, we get Codd's original *difference* operator (MINUS in **Tutorial D**).

39

## Constraints

Constraints express the integrity rules for a database.

Enforcement of constraints by the DBMS ensures that the database is at all times in a *consistent* state.

A constraint is a *truth-valued* expression, such as a *comparison*, declared as part of the *logical schema* of the database.

The comparands of a constraint are typically relation expressions or invocations of aggregate operators.

But the commonest kinds of constraint are expressed using special shorthands, like KEY, FOREIGN KEY, IS_EMPTY.

40

## IS_EMPTY Example

EXAM_MARK

| StudentId | CourseId | Mark |
|---|---|---|
| S1 | C1 | 85 |
| S1 | C2 | 49 |
| S2 | C1 | 49 |
| S3 | C3 | 66 |
| S4 | C1 | 93 |

This might be subject to the constraint:
$0 \leq \text{Mark} \leq 100$

**IS_EMPTY** (
EXAM_MARK WHERE
Mark < 0 OR Mark > 100 )

41

## The End

42