A Short Introduction to Relational Databases

Part 1

Riccardo Murri <riccardo.murri@uzh.ch>

2019-04-09

Databases

"A **database** is an organized collection of data"

"A **database management system** (DBMS) is a computer software application *[...]* A general-purpose DBMS is designed to allow the definition, creation, querying, update, and administration of databases."

— https://en.wikipedia.org/wiki/Database

"A **database** is an organized collection of data"

"A **database management system** (DBMS) is a computer software application *[...]* A general-purpose DBMS is designed to allow the definition, creation, querying, update, and administration of databases."

— https://en.wikipedia.org/wiki/Database

Relational DBs

All major general purpose DBMS's are based on the so-called relational data model.

This means that all data is stored in a number of tables (with named columns), such as:

usr	size	path
usr264	17	/scratch/iftp/usr264/cp1.log
usr116	19362662400	/scratch/id/usr116/vkeller.tar
usr116	3379200	/scratch/id/usr116/test.tar
usr264	16	/scratch/iftp/usr264/cp2.log
usr345	877366	/scratch/aim/usr345/bwa/bwa

For historical and mathematical reasons such tables are referred to as *relations*.

How are relational DBs programmed?

The success of relational databases is largely due to the existence of powerful programming languages for writing database queries.

The most important such language is SQL ("Structured Query Language", sometimes pronounced "sequel").

Relational data model

Relational data model

A **relational database** is a set of relations.

A **relation** is a set of tuples.

A relation can be represented by listing all groups of related elements — the result is a "table".

usr	size	path
usr264	17	/scratch/iftp/usr264/cp1.log
usr116	19362662400	/scratch/id/usr116/vkeller.tar
usr116	3379200	/scratch/id/usr116/test.tar
usr264	16	/scratch/iftp/usr264/cp2.log
usr345	877366	/scratch/aim/usr345/bwa/bwa

Much of the material in these slides comes originally from: *R. Pagh, Introduction to Database Systems,* http://www.itu.dk/people/pagh/IDB05/

What is a tuple?

A relation consists of so-called tuples.

A tuple is an *ordered* list of values.

Tuples are usually written in parentheses, with commas separating the values (or components), e.g.:

```
(Star Wars, 1977, 124, color)
```

which contains the four values `Star Wars`, `1977`, `124`, and `color`.

Note: Order is significant, e.g., the tuple `(Star Wars, 124, 1977, color)` is different from the tuple above.

What is a relation?

A **relation** is a *set* of tuples.

A relation is always defined on certain sets (called *domains*): "the sets from which the values in the tuples come".

Example: The tuple `(Star Wars, 1977, 124, color)` could be part of a relation defined on the sets:

- All text strings,
- the integers,
- the integers, and
- {`'color'`,`'black and white'`}.

However. . .

There are multiple domain sets to which tuple elements could belong.

The tuple `(Star Wars, 1977, 124, color)` could be part of a relation defined on the sets:

- All text strings,
- the integers,
- the integers,
- {`'color'`, `'black and white'`}.

- All text strings,
- the integers,
- floating-point numbers,
- all text strings.

So the domains, over which the components of tuples (belonging to a given relation) are allowed to vary, must be *given*. They are part of a relation's definition.

However. . .

There are multiple domain sets to which tuple elements could belong.

The tuple `(Star Wars, 1977, 124, color)` could be part of a relation defined on the sets:

- All text strings,
- the integers,
- the integers,
- {`'color'`, `'black and white'`}.

- All text strings,
- the integers,
- floating-point numbers,
- all text strings.

So the domains, over which the components of tuples (belonging to a given relation) are allowed to vary, must be *given*. They are part of a relation's definition.

However. . .

There are multiple domain sets to which tuple elements could belong.

The tuple `(Star Wars, 1977, 124, color)` could be part of a relation defined on the sets:

- All text strings,
- the integers,
- the integers,
- `{'color', 'black and white'}`.

- All text strings,
- the integers,
- floating-point numbers,
- all text strings.

So the domains, over which the components of tuples (belonging to a given relation) are allowed to vary, must be *given*. They are part of a relation's definition.

Attributes

In order to be able to refer to the different components in a tuple, we will assign them names (called **attributes**).

Example: For the tuple `(Star Wars, 1977, 124, color)` we might choose the attributes `title`, `year`, `length`, and `filmType`.

Attribute `length` would then refer to the third value of the tuple, `124`.

How do we write relations?

Relations are usually written as two-dimensional tables, with the attributes as a first, special row, and the tuples in the remaining rows.

usr	size	path
usr264	17	/scratch/iftp/usr264/cp1.log
usr116	19362662400	/scratch/id/usr116/vkeller.tar
usr116	3379200	/scratch/id/usr116/test.tar
usr264	16	/scratch/iftp/usr264/cp2.log
usr345	877366	/scratch/aim/usr345/bwa/bwa

Equivalent ways of writing relations

The order of the rows does not matter
(just the *set* of rows).

We may freely reorder the columns,
provided we reorder attributes accordingly.

(This effectively makes tuples a function from the set of attributes into the domains. Defining tuples as ordered list of values was just a preliminary step to get here.)

Equivalent ways of writing relations

The order of the rows does not matter
(just the *set* of rows).

We may freely reorder the columns,
provided we reorder attributes accordingly.

(This effectively makes tuples a function from the set of attributes into the domains. Defining tuples as ordered list of values was just a preliminary step to get here.)

Recap: A formal definition

Let A be a finite set. It is the set of *attributes*.

Let $D : A \to \mathbf{Set}$ be a map from A into the set of (finite) sets, i.e., for every $a \in A$ we are given a finite set D_a the *domain* of attribute a. D is the relation *schema*.

A *tuple* is an element of the product set $\prod_{a \in A} D_a$, i.e., the choice, for every $a \in A$, of an element $t(a) \in D_a$.

A *relation* is a set of tuples, i.e., a subset of $\prod_{a \in A} D_a$.

A (relational) *database* is a set of relations.

Basic SQL

Naming of parts

Rel. Algebra	$\Leftrightarrow$	**SQL DBMS**
attributes		fields, column names
domain		column type
relation		table, view*
tuple		row, record

* There is a difference between “table” and “view” in SQL, to be explained later.

The SQL `SELECT` statement

```
db=> SELECT 'hello, world!';
   ?column?
---------------
 hello, world!
(1 row)
```

The `SELECT` statement is used to compute expressions in SQL.

The result of evaluating a `SELECT` statement is again a *relation* (table).

The SQL SELECT statement

```
db=> SELECT 'hello, world!';
   ?column?
---------------
 hello, world!
(1 row)
```

The SELECT statement is used to compute expressions in SQL.

The result of evaluating a SELECT statement is again a *relation* (table).

Why is this important?

The SQL `SELECT` statement

```
db=> SELECT 'hello, world!';
   ?column?
---------------
 hello, world!
(1 row)
```

The `SELECT` statement is used to compute expressions in SQL.

The result of evaluating a `SELECT` statement is again a *relation* (table).

It allows *composition* of queries into larger expressions.

What is an algebra?

An algebra consists of a set of **atomic operands**, and a set of **operators**.

We can form algebraic expressions by applying operators to operands (which can be atomic or expressions themselves).

For this to work, the result of applying an operator to an expression must be again an operand. In other words, an algebra must be *closed* under all given operations.

What is *relational* algebra?

Relational algebra, defined in its basic form by E. F. Codd in 1970, has:

- relations as atomic operands, and
- various operations on relations as operators (which will be detailed shortly).

Relational algebra is the basis of SQL.

Basic SQL syntax

```
db=> SELECT 'hello, world!';
   ?column?
---------------
 hello, world!
(1 row)
```

Strings in SQL are delimited by *single* quotes.

The SQL `SELECT` statement

```
db=> SELECT 'hello, world!';
    ?column?
---------------
 hello, world!
(1 row)
```

Columns in a table must have a name, so the DBMS makes up one.

The SQL SELECT statement

```
db=> SELECT 'hello, world!'
db->        AS text  ;
    text
---------------
 hello, world!
(1 row)
```

A column name can be given with the AS clause.

The SQL `SELECT` statement

```
db=> SELECT 'foo' , 'bar';
 ?column? | ?column?
----------+----------
 foo      | bar
(1 row)
```

Multiple expressions can be evaluated by separating them with a comma.

They define different columns in the result table.

NULL and UNKNOWN

NULL is a special value that may be used for any data type when no other value is applicable.

Evaluating expressions involving NULL:

- Arithmetic expressions involving NULL evaluate to NULL.
- Boolean expressions evaluate to NULL only when the correct value cannot be deduced.

Some RDBMSs support a special null value for booleans called UNKNOWN.

Selection

`SELECT ... FROM ... WHERE` returns a relation of all rows in a table that satisfy a certain predicate.

```
db=> SELECT *
db-> FROM lustre
db-> WHERE usr='usr116';
  usr    |...|    size     | path
---------+---+-------------+-------------------------------
 usr116  |...| 256958166   | /scratch/id/usr116/pic_babo.tgz
 usr116  |...|   2101760   | /scratch/id/usr116/testib.tar
 [...]
```

Selection

`SELECT ... FROM ... WHERE` returns a relation of all rows in a table that satisfy a certain predicate.

```
db=> SELECT *
db-> FROM lustre
db-> WHERE usr='usr116';
  usr    |...|    size     | path
---------+---+-------------+-------------------------------
 usr116  |...|  256958166  | /scratch/id/usr116/pic_babo.tgz
 usr116  |...|    2101760  | /scratch/id/usr116/testib.tar
 [...]
```

$\star$ is a shorthand for "all column names".

Selection

`SELECT ... FROM ... WHERE` returns a relation of all rows in a table that satisfy a certain predicate.

```
db=> SELECT *
db-> FROM lustre
db-> WHERE usr='usr116';
  usr    |...|    size     | path
---------+---+-------------+-------------------------------
 usr116 |...| 256958166 | /scratch/id/usr116/pic_babo.tgz
 usr116 |...|   2101760 | /scratch/id/usr116/testib.tar
 [...]
```

`FROM` indicates the table from which to extract rows.

Each table in a SQL database is given a name (at creation time).

Selection

`SELECT ... FROM ... WHERE` returns a relation of all rows in a table that satisfy a certain predicate.

```
db=> SELECT *
db-> FROM lustre
db-> WHERE usr='usr116';
  usr    |...|    size     | path
---------+---+-------------+-------------------------------
 usr116 |...| 256958166 | /scratch/id/usr116/pic_babo.tgz
 usr116 |...|   2101760 | /scratch/id/usr116/testib.tar
 [...]
```

The `WHERE` clause specifies the predicate that selected rows must satisfy.

Selection

`SELECT ... FROM ... WHERE` returns a relation of all rows in a table that satisfy a certain predicate

```
db=> SELECT *
db-> FROM lustre
db-> WHERE usr ='usr116';
  usr    |...|    size     | path
---------+---+-------------+-------------------------------
 usr116  |...| 256958166   | /scratch/id/usr116/pic_babo.tgz
 usr116  |...|   2101760   | /scratch/id/usr116/testib.tar
 [...]
```

The `WHERE` clause specifies the predicate that selected rows must satisfy.

Column names are used in predicate expression to denote the value of the corresponding attribute value in a tuple.

Selection

`SELECT ... FROM ... WHERE` returns a relation of all rows in a table that satisfy a certain predicate

```
db=> SELECT *
db-> FROM lustre
db-> WHERE usr = 'usr116';
  usr    |...|    size     | path
---------+---+-------------+-------------------------------
 usr116  |...|  256958166  | /scratch/id/usr116/pic_babo.tgz
 usr116  |...|    2101760  | /scratch/id/usr116/testib.tar
 [...]
```

The `WHERE` clause specifies the predicate that selected rows must satisfy.

If the `WHERE` clause is omitted, all rows are selected.

Predicates in `WHERE` clauses

SQL offers a variety of ways of forming conditional expressions in `WHERE` clauses:

- Comparison operators (used between e.g. a pair of integers or strings): `=`, `<`, `<=`, `>`, `>=`, `<>`.
- Boolean operators (used to combine conditional expressions): `AND`, `OR`, `NOT`.
- The `LIKE` operator used to find strings that match a given pattern.
- Parentheses can be used to indicate the order of evaluation. If no parentheses are present, a standard order is used.

Selection

`SELECT ... FROM ... WHERE` returns a relation of all rows in a table that satisfy a certain predicate

```
db=> SELECT *
db-> FROM lustre
db-> WHERE LENGTH(path) <32;
  usr    |...|    size     | path
---------+---+-------------+--------------------------------
 usr116 |...| 256958166 | /scratch/id/usr116/pic_babo.tgz
 usr116 |...|   2101760 | /scratch/id/usr116/testib.tar
 [...]
```

The `WHERE` clause specifies the predicate that selected rows must satisfy.

Functions can be applied to column values.

RDBMS provide some built-in functions but generally let users also define their own (UDF).

Projection

A list of column names or expressions can be given to `SELECT`; the resulting relation will contain only the specified columns.

```
db=> SELECT usr, size, path
db-> FROM lustre;
  usr   |  size  |                path
--------+--------+-----------------------------------
 us345  |    100 | /scratch/aim/bin/run_fastqc.bat
 us345  | 507539 | /scratch/aim/bin/sam-1.32.jar
 [...]
```

Set vs. *Bag* semantics

A list of column names or expressions can be given to `SELECT`; the resulting relation will contain only the specified columns.

```
db=> SELECT usr
db-> FROM lustre;
  usr
-------
 usr345
 usr345
 usr345
 [...]
```

However, selecting a subset of attributes may lead to *non-unique* tuples.

Set vs. *Bag* semantics

SQL defines a relation as a *list* of tuples,
i.e., an ordered sequence of rows,
allowing identical tuples to appear.

The reason is *efficiency*: eliminating duplicates is (computationally) costly, so is only done at the programmers' request.

DISTINCT

The `DISTINCT` modifier is used to force removal of duplicate values in an expression:

```
db=> SELECT DISTINCT usr
db-> FROM lustre;
  usr
-------
 usr345
 usr116
 [...]
```

Keys

A subset A of the attributes of relation R such that there are no duplicates when R is projected upon A is called a **key**.

```
db=> SELECT path FROM lustre;
 [...]
(30356776 rows)
```

```
db=> SELECT DISTINCT path FROM lustre;
 [...]
(30356776 rows)
```

Keys

A subset A of the attributes of relation R such that there are no duplicates when R is projected upon A is called a **key**.

The uniqueness constraints of the key must hold however the relation is updated and modified.
In other words, a key constraint is an existential attribute of the relation, not just an accident of the data.

So, key constraints are specified when the table is created.

Ordered relations, again

SQL defines a relation as a *list* of tuples, i.e., an ordered sequence of rows, allowing identical tuples to appear.

Can you think of another reason why SQL defines a relation as an *ordered* sequence?

(Answer on next slide.)

Sorting

An additional clause `ORDER BY` allows sorting on an arbitrary expression of the attribute values.

```
db=> SELECT DISTINCT usr
db-> FROM lustre
db-> ORDER BY usr ASC
db-> LIMIT 3;
  usr
-------
 us293
 us319
 us320
(3 rows)
```

Sorting

An additional clause `ORDER BY` allows sorting on an arbitrary expression of the attribute values.

```
db=> SELECT DISTINCT usr
db-> FROM lustre
db-> ORDER BY usr ASC
db-> LIMIT 3;
  usr
-------
 us293
 us319
 us320
(3 rows)
```

Any valid expression can be used as a sorting key!

Sorting

An additional clause `ORDER BY` allows sorting on an arbitrary expression of the attribute values.

```
db=> SELECT DISTINCT usr
db-> FROM lustre
db-> ORDER BY usr ASC
db-> LIMIT 3;
  usr
-------
 us293
 us319
 us320
(3 rows)
```

Specify **ascending** sort order.

Use `DESC` for the opposite ordering.

Aside: LIMIT

An additional clause `LIMIT` allows specifying a maximum size of the result set. It can be applied to *any* SQL query.

```
db=> SELECT DISTINCT usr
db-> FROM lustre
db-> ORDER BY usr ASC
db-> LIMIT 3;
  usr
-------
 us293
 us319
 us320
(3 rows)
```

Aggregate functions

Computing on aggregates

Some functions by definition operate on a group of rows and return a single value out of the group:

```
db=> SELECT AVG(size) FROM lustre;
          avg
-----------------------
 4815351.200747734213

db=> SELECT MAX(size) FROM lustre;
     max
--------------
  354519221688

db=> SELECT COUNT(*) FROM lustre;
  count
----------
 30356776
```

Computing on aggregates

By default, aggregation is over all selected rows.

A `GROUP BY` clause can be used to specify how rows should be grouped; expressions involving aggregate functions will be computed once per each group.

```
db=> SELECT usr, MAX(size)
db-> FROM lustre
db-> GROUP BY usr;
  usr    |      max
---------+---------------
 usr25   | 354519221688
 usr324  |   3026168420
 usr234  |   2038075132
 ...
```

Computing on aggregates

An additional `HAVING` clause applies a predicate to further select groups based on some expression.

For example, compute average file size per user, but only consider users that have at least 10000 files on the system:

```
db=> SELECT usr, AVG(size)
db-> FROM lustre
db-> GROUP BY usr
db-> HAVING COUNT(path)>10000;
  usr    |          avg
---------+-----------------------
 usr25   | 19942289.171742225897
 usr324  |  1891531.025947844280
 usr234  |  1443121.015355730924
 ...
```

Computing on aggregates

For example, compute average file size per user, but only consider users that have at least 10000 files on the system:

```
db=> SELECT usr, AVG(size)
db-> FROM lustre
db-> GROUP BY usr
db-> HAVING COUNT(path)>10000;
  usr    |          avg
---------+-----------------------
 usr25   | 19942289.171742225897
 usr324  |  1891531.025947844280
 usr234  |  1443121.015355730924
 ...
```

Could this be done with a `WHERE` clause? Why?

Subqueries

Subqueries

Until now, we have seen SQL queries of the form:

```
SELECT list of attributes
FROM list of relations
WHERE condition;
```

What we haven't used is that:

- In any place where a relation is allowed, we may put an SQL query (a subquery) computing a relation.
- In any place where an "atomic value" (e.g. string or integer) is allowed, we may put an SQL subquery computing a relation with one value of this type (i.e., having one attribute and one tuple).

Subqueries in the `FROM` clause

Instead of just relations, we may use SQL queries in the `FROM` clause of a `SELECT` statement.

If we need a name for referring to the relation computed by the subquery, an additional `AS` *`name`* is appended after the subquery. This allows us to use values from the subquery results in other expressions.

Subqueries should always be surrounded by parentheses.

Subqueries producing scalar values

When a query produces a relation with one attribute and one tuple, it can be used in any place where we can put an atomic (or scalar) value.

In places where an atomic value is expected, SQL regards a relation instance containing one atomic value x to be the same as the value x.

If such a subquery does not result in exactly one tuple, it is a run-time error, and the SQL query cannot be completed.

Subqueries in conditions

One common use of subqueries is in the `WHERE` part of a `SELECT` statememt.

There are several operators in SQL that apply to a relation S, result of a subquery, and produce a boolean result:

- `EXISTS` S is true if and only if S is not empty.
- t `IN` S is true if and only if t is a tuple in S.

If S is unary (has just one attribute):

- x `> ALL` S is true if and only if x is greater than all values in S.
- x `> ANY` S is true if and only if x is greater than some value in S

(and similarly for other comparison operators).

Appendix

References

Pagh, R.: “Introduction to Database Systems”, http://www.itu.dk/people/pagh/IDB05/

Codd, E. F.: “A relational model for large shared data banks”, *Comm. ACM* 13:6, pp. 377–387, 1970