CS2102

Database Systems

AY2022/23 Semester 2

Notes by Jonathan Tay

Last updated on January 24, 2023

Contents

Part I

Relational Model

Data in **relational databases** are stored in **relations** (tables). Column headers are **attributes**, and rows are **tuples**.

The **degree** is the number of columns, and the **cardinality** is the number of rows.

The **domain** of an attribute A_i, denoted as $\text{dom}(A_i)$, is the set of all possible *atomic* values for A_i. `NULL` is an additional special value for unknown or invalid values.

keys

A **superkey** is a subset of attributes that uniquely identifies a tuple. A **key** is a *minimal* superkey.

The **candidate keys** is the set of all keys for a relation, of which one is selected as a **primary key**.

Primary key values must be non-`NULL`.

foreign keys

A **foreign key** is a subset of attributes of a *referencing relation* that refers to the primary key of a *referenced relation*:

(referencing attributes) ⇝ (referenced attributes)

Because the names of the attributes are not necessarily unique, each attribute is prefixed with the name of the relation, like so:

(<relation name> · <attribute name>, ...) ⇝ ...

Foreign keys must appear as a primary key in the referenced table, `NULL`, or a tuple containing `NULL`.

The key constraints above are not intrisic properties of a relation; rather, they are specified by the database designer to avoid problematic but otherwise valid data.

1 Relational Algebra

Relation are always the **operands** in relational algebra, on which **operators** are applied.

Because relations are closed under *any combination of operators*, the result of an operation is *always* a relation, and no other output is possible (**closure property**).

3-valued logic (`false`, `true`, and `NULL`)

Any operation involving `NULL` will result in `NULL`. Hence, ≡ and ≢ are needed to compare (in)equality of `NULL` values directly:

`NULL = NULL` produces `NULL`, but `NULL ≡ NULL` produces `true`.

$a \wedge b$		a		
		true	**false**	**NULL**
b	**true**	true	false	NULL
	false	false	false	false
	NULL	NULL	false	NULL

$a \vee b$		a		
		true	**false**	**NULL**
b	**true**	true	true	true
	false	true	false	NULL
	NULL	true	NULL	NULL

1.1 Equivalence and Compatibility

Two *expressions* are **equivalent** if either *both* produce an error, or *both* produce the same result.

Errors occur if attributes are missing (e.g. by projection or renaming), or are incompatible — there is no implicit type conversion.

The order of types in a tuple matters — (`int`, `text`) is not equivalent to (`text`, `int`).

Two *relations* are **union-compatible** if they have the same number of attributes with the same domain (type).

1.2 Basic Operators

Note that logical conjunction (∧) has greater precedence than logical disjunction (∨).

selection — $\sigma_{[c]}(R)$

Filters the rows of relation R, returning the set of tuples that satisfy the condition c.

Conditions which evaluate to `NULL` are excluded from the result.

The condition c must only specify attributes of R.

projection — $\pi_{[l]}(R)$

Maps the relation R, returning the set of tuples with the attributes in the ordered list l, in the order specified by l.

Equivalent to a column filter, but the rows which were previously unique tuples may no longer be unique, and are therefore de-duplicated.

The elements of l must not be operations, must be attributes of R, and must be unique.

renaming — $\rho_{[\mathcal{R}]}(R)$

Renames the attributes of relation R to the attributes in the unordered list $\mathcal{R}$.

Elements of $\mathcal{R}$ must be in the following form: <new name> ← <old name>.

New attribute names must be unique, and existing attributes must only be renamed at most once per operation (i.e. $\rho_{[A \leftarrow B, B \leftarrow C]}$ is invalid).

1.3 Set Operators

The typical set union (∪), set intersection (∩), and set difference (—) operators are omitted for brevity.

cross product — $R \times S$

For every tuple in R, concatenate it with every tuple in S to form a new relation, such that the cardinality (number of rows) of the result is $|R| \times |S|$.

The set of attributes in R and S must be disjoint, such that the degree (number of columns) of the result is $\deg(R) + \deg(S)$.

Cross products are associative — $R \times (S \times T) = (R \times S) \times T$.

1.4 Join Operators

Joins are a composite operator, composing cross product, selection, and projection on a relation.

This concatenates two tables and removes unwanted/redundant rows and columns from the result of a cross product.

theta-join (θ-join) — $R \bowtie_{[\theta]} S$

Cross R and S, then filter (by selection) the rows using the condition θ.

equi-join — $R \bowtie_{=} S$

A special case of theta-join, where only equality (=) conditions are allowed (c.f. θ-join which allows ≡, ≤, <>, etc.).

This may be more performant versus a theta-join as hashing can be used internally.

natural inner join — $R \bowtie S$

First, find the set of attributes that are common to both R and S.

Then, cross R and S, and retain (by selection) the rows for which the **common attributes** are equal — this also eliminates rows with any `NULL` value.

If there are no common attributes, then this is simply the cross product (by vacuous truth).

Finally, de-duplicate (by projection) the columns by their attribute names.

Theta-joins, equi-joins, and natural inner joins are collectively known as **inner joins**.

If we wish to perform a join but still retain *all rows* from the *left* table, *right* table, or even *both* tables and simply pad missing values with `NULL`, we can use **outer joins**.

full outer join — $L \overline{\bowtie}_{[\theta]} R$

attributes of L		attributes of R	
values from L	⋯	values from R	…
⋮	⋱	⋮	⋱
values from L	…	`NULL`	…
⋮	⋱	⋮	⋱
`NULL`	…	values from R	…
⋮	⋱	⋮	⋱

Inner joins only retain rows (in red) from which both L and R have a value for which the condition θ evaluates to `true` (during the selection).

However, rows (in green and blue) which do not satisfy θ may also be desirable, and can be retained by outer joins.

The **full outer join** retains all of the rows above.

left outer join — $L \overline{\bowtie}_{[\theta]} R$

The left outer join retains the red and green rows.

right outer join — $L \overline{\bowtie}_{[\theta]} R$

The right outer join retains the red and blue rows.

natural outer joins

The natural keyword can be prefixed to the left, right, or full outer joins, e.g. "natural left outer join".

The equality condition is implicitly defined over the set of attributes that are common to both L and R.

Part II

SQL

SQL (Structured Query Language) is case-insensitive, but keywords are uppercase by convention.

The most common data types are `INTEGER`, `VARCHAR(n)`, `BOOLEAN`, and `DATE`.

2 Operations and Syntax

2.1 Table Operations

```
CREATE TABLE table_name (
   -- define attributes (columns)
   <attribute> <type> [<column_constraint>],
   <attribute> <type> [CONSTRAINT <name> <cstr.>],
   ...
   -- define optional table constraints
   [<table_constraint>],
   -- constraints can be named
   [CONSTRAINT <name> <table_constraint>],
   ... /* alternative comment syntax */
   [<table_constraint>] -- no trailing comma
);

ALTER TABLE <table>
   [ALTER / ADD / DROP] [COLUMN / CONSTRAINT]
   <attribute / name>
   <changes>;

DROP TABLE
   [IF EXISTS] -- no error if table doesn't exist
   <table>, ... -- drop multiple tables at once
   [CASCADE]; -- also drop referencing tables
```

2.2 Integrity Constraints

Constraints are specified in the `CREATE TABLE` statement, and reject insertions if the condition evaluates to `false` (**principle of rejection**).

Column constraints are defined on a per-column basis, while **table constraints** are defined on the table as a whole.

Constraints can be *named*, or *unnamed*, in which case the DBMS will generate a name.

There are 5 types of constraints that can be placed on a table R and/or any attribute a:

1. **`NOT NULL`**: $\forall r \in R : r_a \not\equiv \texttt{NULL}$
2. **`UNIQUE`**: $\forall r_1, r_2 \in R : (r_1 \equiv r_2) \vee (\exists r_{1_a} <> r_{2_a})$
3. **`PRIMARY KEY`**: equivalent to `UNIQUE` and `NOT NULL`
4. **`FOREIGN KEY (`**$a, \ldots$**`) REFERENCES`** R'**`(`**$a', \ldots$**`)`**: $\forall r \in R : (\forall a : r_a \in R'_{a'}) \vee (\texttt{NULL} \in r)$
5. **`CHECK(`**c**`)`**: c does not evaluate to `false`

For `UNIQUE`, note that `NULL <> NULL` evaluates to `NULL`, such that duplicate insertions of `NULL` are not rejected.

For `FOREIGN KEY`, R' must be a valid table name. `SET NULL`, `SET DEFAULT`, or `CASCADE` can be specified as the action to take when a referenced row is deleted.

`CASCADE` will delete all referencing rows, propagating the deletion, which may significantly affect performance.

For `CHECK`, c must be a boolean expression scoped to the table on individual rows.

2.3 Row Operations

```
INSERT INTO <table> [(attribute, ...)]
   VALUES -- whitespace and newlines optional
      (value, ...),
      (value, ...),
      ... -- alternatively, all on one line
      (value, ...);
```

All values to insert must have the same shape as either the table, or, if specified, the attribute list.

Attributes in the attribute list can be in any order.

Attributes missing from the attribute list will have their values set to `NULL`, or, if specified in the schema, to their default value.

```
DELETE FROM <table> [WHERE <condition>];
```

The condition only deletes rows which evaluate to TRUE (**principle of acceptance**).

If the condition is omitted, all rows will be deleted.

The condition must be a boolean expression scoped to to the table on individual rows.

2.4 Deferrable Constraints

```
BEGIN; -- start a transaction
-- perform operations
COMMIT; -- commit (end) the transaction
```

Constraints are checked immediately at the end of every SQL statement (these end with a semicolon) and transaction — violation performs a rollback.

When defining a constraint, three deferments can be specified:

1. **`NOT DEFERRABLE`**:
   if unspecified, this is the default
2. **`DEFFERABLE INITIALLY IMMEDIATE`**:
   constraint is checked immediately, but can be deferred later
3. **`DEFFERABLE INITIALLY DEFERRED`**:
   constraint is deferred by default

`DEFFERABLE INITIALLY IMMEDIATE` gives the *option* to defer the constraint checking by adding the following line in a transaction, i.e.:

```
BEGIN;
SET CONSTRAINTS <name> DEFERRED; -- add this line
-- perform operations
COMMIT;
```

Deferring a constraint is necessary when the constraint depends on a row that is inserted later in the transaction, e.g. circular foreign key constraints.