CS2102

Database Systems

AY2022/23 Semester 2

Notes by Jonathan Tay

Last updated on April 1, 2023

Contents

Part I

Relational and ER Models

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.

2 Entity Relationship Model

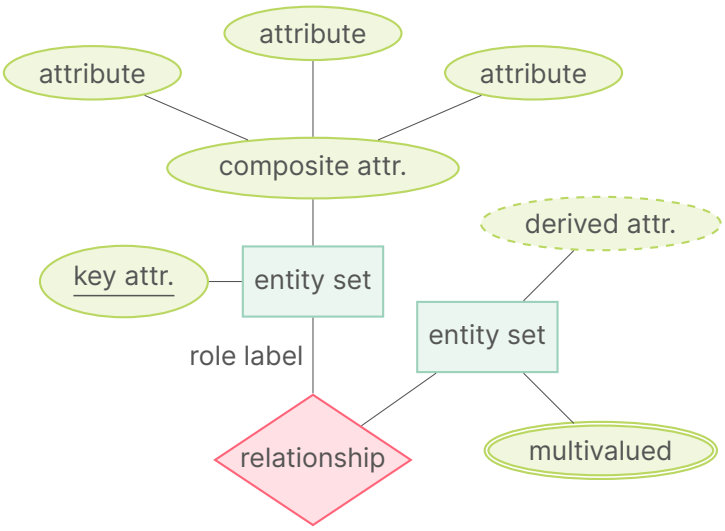
In the ER model, data is described as a collection of:

- **entities**:
  representation of real-world objects that are distinguishable from other objects
- **relationships**:
  associations between one or more entities

Entities and relationships of the same type form **entity sets** and **relationship sets** respectively.

Attributes describe information about entities and relationships, and there are various kinds:

- **key**: uniquely identify an entity
- **composite**: collection of multiple *attributes*
- **multivalued**: collection of multiple possible *values*
- **derived**: calculated from other attributes



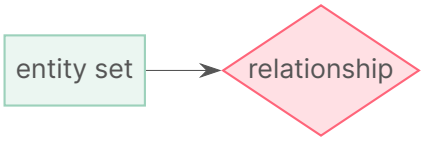
The **degree** (or *arity*) of a relationship set is the number of entity sets involved — binary for $n = 2$, ternary for $n = 3$.

2.1 Relationship Constraints

cardinality constraints

Relationships can be **many-to-many**, **many-to-one**, or **one-to-one**. These are *upper bounds*.

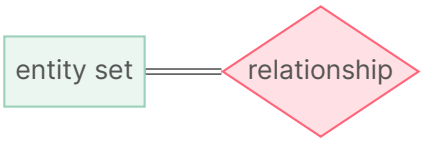
The default upper limit is ∞, and setting an upper limit of 1 is depicted in ER diagrams by an arrowhead:



participation constraints

Participation constraints can be **partial** (0 or more) or **total** (1 or more). These are *lower bounds*.

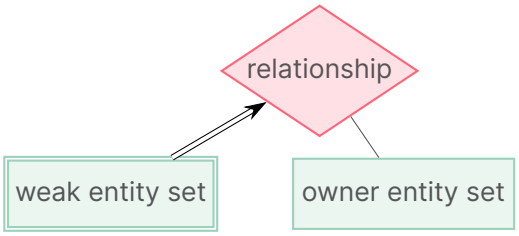
The default lower limit is 0, and setting a lower limit of 1 is depicted in ER diagrams by a double line:



dependency constraint

A **weak entity set** does not have its own key, instead having a **partial key** which can only uniquely identify entities with a primary key from its **owner entity set**.

This requires a connection via an **identifying relationship set**:



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`.

3 Operations and Syntax

3.1 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
```

3.2 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.

3.3 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.

3.4 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.

4 Queries

SQL by default does not eliminate duplicate rows (without DISTINCT) nor does it have a fixed row ordering (**order-independent**) (without ORDER BY).

The basic query syntax is as follows:

```
SELECT [DISTINCT] <attributes> -- * or a1, a2, ...
   FROM <tables> -- r1, r2, ...
   WHERE <conditions> -- c
   ORDER BY <attributes> <ASC / DESC>
   OFFSET <n>
   LIMIT <n>; -- ; only if nothing else follows
[[UNION / INTERSECT / EXCEPT] [ALL]]
   ... -- more queries
```

DISTINCT uses IS DISTINCT FROM to compare rows, which is different from =. The latter returns NULL if either argument is NULL, while the former treats NULL like any other value.

Condition evaluation is based on the **principle of acceptance** — rows are included if and only if the condition evaluates to TRUE.

The above query up to line 3 is equivalent to the following relational algebra expression:

$$\pi_{[a_1,a_2,\dots]}\left(\sigma_{[c]}(r_1 \times r_2 \times \dots)\right)$$

SELECT and FROM clauses — AS expression

Rename columns or tables (within the query scope) with AS, and even operate on entire columns with several functions:

- **mathematical**: + - * / % |/ ^
- **logical**: AND, OR, NOT
- **string**: ||, LOWER(s), UPPER(s)
- **date**: +, NOW()

```
SELECT <regular attributes> -- a1, a2, ...
   [<expression> AS <new attribute>], -- ...
FROM <regular tables>,
   [<old table name> [AS] <new table name>];
```

Renaming tables is necessary if the same table appears more than once in the FROM clause.

The AS keyword is optional when used in FROM clauses.

WHERE clause

Pattern match (or with regex) with LIKE and NOT LIKE as a condition:

- an underscore (_) matches exactly 1 character,
- % matches 0 or more characters.

```
SELECT <attributes>
FROM <tables>
WHERE <attribute> [[NOT] LIKE '<pattern>'];
```

The logical operators AND, OR, and NOT can be used to combine/invert conditions.

4.1 Set Operations

Without ALL, duplicate rows are eliminated after UNION, INTERSECT, and EXCEPT (set difference).

With ALL, for every row x in tables A and B appearing a and b times respectively, x will appear these many times in the result:

- **UNION**: $a + b$
- **INTERSECT**: $\min(a, b)$
- **EXCEPT**: $\max(a - b, 0)$

4.2 Join Operations

```
SELECT <attributes>
FROM <r1> [NATURAL] [LEFT / RIGHT / FULL] JOIN
     <r2> [ON <condition>],
... -- more joins, query continues
```

If the condition and ON keyword are omitted, the join is implicitly a natural inner or outer join, and the NATURAL keyword is optional.

4.3 Composition

A **scalar subquery** is a query that returns a single value (a table with 1 row and 1 column), or an empty table which is treated as NULL.

They are dynamically checked at runtime and can be used as a value in a SELECT, FROM, or WHERE clause.

There are other types of subqueries for dynamically checking if values are in a set or not:

```
...
WHERE -- subsequent lines are mutually exclusive
   <expr.> [NOT] IN (<subquery / values>);
   <expr.> <op.> ALL (<subquery / values>);
   <expr.> <op.> ANY (<subquery / values>);
```

For ALL and ANY, op can be any comparison operator like =, <, >, <=, >=, <>, while IN uses = and <> implicitly (not IS DISTINCT FROM!).

The subquery must return exactly one column for all the above.

There is also WHERE [NOT] EXISTS (<subquery>), which is true if the subquery returns at least one row, which can lead to unusual queries like this:

```
...
WHERE EXISTS (
   SELECT 1 -- any value will do
   FROM <table>
   WHERE <condition>
);
```

Unlike the others, the subquery of EXISTS does not need to return a single column.

5 Persistent Stored Modules

5.1 Functions

Functions in SQL are used to perform operations on data, and must return at least one value.

```
CREATE OR REPLACE FUNCTION <name> (<params>)
   RETURNS <return type> AS $$
      <body>
      [LIMIT 1;] -- for RECORDs
   $$ LANGUAGE sql;
```

Input parameters (constants) are prefixed with IN, output parameters (uninitialized return values) with OUT, and input/output parameters (initialized variables) with INOUT.

The return type can be any arbitrary type, but returning an existing tuple requires the table name as the return type.

However, if the function returns a new tuple, the return type must be a RECORD or SETOF RECORD, or equivalently, TABLE.

Function bodies cannot commit or rollback transactions.

Functions are called using regular SELECT statements:

```
SELECT <name>(<parameters>);
```

function variables

Variables can be declared in function bodies, and they are local to the function.

```
   -- ... $$
   DECLARE <variable name> <type> [:= <value>];
   BEGIN <body> END;
   -- $$ ...
```

function control flow

If statements and loops are available as well.

```
-- ... $$
   IF <condition> THEN <body>
   [ELSEIF <condition> THEN <body>]
   [ELSE <body>]
   END IF;
   LOOP [EXIT WHEN <condition>;]
      <body>;
   END LOOP;
-- $$ ...
```

cursors

Cursors are used to iterate over the result of a query.

```
-- $$ ...
   DECLARE <cursor name> CURSOR FOR <query>;
   r RECORD;
   BEGIN
```

```
    OPEN <cursor name>;
    LOOP
       FETCH <cursor name> INTO r;
       EXIT WHEN NOT FOUND;
       <body>;
       [RETURN NEXT;]
    END LOOP;
    CLOSE <cursor name>;
END
-- $$ ...
```

`RETURN NEXT;` is used to insert the current value of `r` into the result of the function.

The function return type must be `SETOF RECORD` or `TABLE` for this to work.

Variants of `FETCH <variant> FROM` with `PRIOR`, `FIRST`, `LAST`, `ABSOLUTE <n>` and `RELATIVE <n>` are also available.

5.2 Procedures

Unlike functions, procedures do not return any values, but they can commit or rollback transactions as long as they are not called within a transaction.

Procedures have access to the same variable, control flow and cursor constructs as functions.

```
CREATE OR REPLACE PROCEDURE <name> (<params>)
   AS $$
      <body>
   $$ LANGUAGE sql;
```

Since functions do not return values, their parameters are implicitly (and need not be) prefixed with `IN`. Nonetheless, `INOUT` can be used too.

Procedures are called with the `CALL` keyword:

```
CALL <name>(<parameters>);
```

5.3 Triggers

Triggers are used to execute **trigger functions** when a certain event occurs.

As the name implies, trigger functions are functions with a return type of `TRIGGER`.

To get the database to execute a trigger function, it must be registered as a trigger:

```
CREATE TRIGGER <name>
   -- trigger events, can use multiple with OR
   [<AFTER / BEFORE / INSTEAD OF> -- timing
      <INSERT / UPDATE / DELETE>] -- operation
   ON <table>
   [FOR EACH <ROW / STATEMENT>] -- level
   [WHEN <condition>]
   EXECUTE FUNCTION <function name>(<params>);
```

Trigger functions can access the inserted or updated tuple using the `NEW` variable, and the replaced or deleted tuple using `OLD`.

Other special keywords such as `TG_OP` to get the operation type (e.g. `INSERT`) and `TG_TABLE_NAME`.

trigger levels

`FOR EACH ROW` invokes the trigger function for each row affected by the operation.

`FOR EACH STATEMENT` is used to perform operations on the entire table, and the trigger function is invoked once per operation.

This means that statement-level triggers cannot access the `NEW` and `OLD` variables.

Neither can the statement's operation be prevented by returning `NULL`, unless an exception is raised with `RAISE EXCEPTION`.

`RAISE NOTICE <message>` can be used to print a message to the console, but is not equivalent to `RAISE EXCEPTION <message>`.

trigger timing - BEFORE

Returning `NULL` will prevent the operation (insertion, updating, deletion) from happening.

Since the value of `OLD` is initialized to `NULL`, `RETURN OLD` will also prevent the operation from happening *unless* some columns of `OLD` are modified.

trigger timing - AFTER

The return value of the trigger function has **no effect** on the operation as it is invoked after the operation has already happened.

trigger timing - INSTEAD OF (row-level only)

This may only be defined on views, intended to perform the operation on the underlying table instead of the view.

Like `BEFORE`, returning `NULL` will prevent the operation from happening.

Triggers are checked after every statement immediately by default, but can be deferred until after the `COMMIT` of a transaction.

```
CREATE CONSTRAINT TRIGGER <name>
    AFTER <operation> ON <table>
    FOR EACH ROW
    DEFERRABLE INITIALLY <DEFERRED / IMMEDIATE>
    EXECUTE FUNCTION <function name>(<params>);
```

Deferred triggers must be `AFTER` operations `FOR EACH ROW`.

Since triggers are constraints, any deferrable triggers which are set to `INITIALLY IMMEDAITE` can be disabled on a per-transaction basis by changing the trigger constraint.

```
BEGIN TRANSACTION;
SET CONSTRAINTS <trigger name> DEFERRED;
-- ...
COMMIT;
-- trigger is deferred until here
```

If there are multiple triggers on the same operation of a table, the following order applies, ties broken by alphabetical order of the trigger name:

1. `BEFORE ... FOR EACH STATEMENT` triggers
2. `BEFORE ... FOR EACH ROW` triggers
3. `AFTER ... FOR EACH ROW` triggers
4. `AFTER ... FOR EACH STATEMENT` triggers

If a row-level `BEFORE` trigger function returns `NULL`, then all subsequent triggers on the same row are skipped.

Part III

Normal Forms

A **normal form** is a definition of minimum requirements to reduce data redundancy and improve data integrity.

Redundancies can lead to **anomalies** where a primary key attribute violates the constraint of being non-`NULL`.

These anomalies can be resolved by **normalization**.

6 Functional Dependencies

A **functional dependency** is a relationship between two sets of attributes A and B that occur when the value of B can always be determined from the value of A.

That is, when $A_1A_2 \ldots A_m \rightarrow B_1B_2 \ldots B_n$, any tuple with the same values for $A_1A_2 \ldots A_m$ will have the same value for $B_1B_2 \ldots B_n$.

determining functional dependencies

Intuitively, the $\rightarrow$ can be read as "decides" or "determines".

Given attributes A and B, if A is a key of the relation (all unique values), then $A \rightarrow B$.

Another method is to construct a counter-example with 2 tuples which would violate the functional dependency.

The attribute with the same value in all rows will be on the LHS of FD, and the other attribute on the RHS.

We can use **Armstrong's axioms** and some additional rules (denoted with †) to determine if a functional dependency is valid.

- **augmentation**: if $A \rightarrow B$ then $AC \rightarrow BC$ for any C
- **transitivity**: if $A \rightarrow B$ and $B \rightarrow C$ then $A \rightarrow C$
- **reflexivity**: any set of attributes $\rightarrow$ subset of the attributes
- **decomposition**†: if $A \rightarrow BC$ then $A \rightarrow B$ and $A \rightarrow C$
- **union**†: if $A \rightarrow B$ and $A \rightarrow C$ then $A \rightarrow BC$

Using these rules is pretty tedious, and we can use **closures** to simplify the process.

Given a set of functional dependencies, a closure $\{A_1, A_2, \ldots, A_n\}^+$ is the set of all attributes that can be determined from $\{A_1, A_2, \ldots, A_n\}$.

proving functional dependencies $A \rightarrow B$

To prove that $A \rightarrow B$ holds, we must show that $B \in \{A\}^+$.

To prove that $A \rightarrow B$ does not hold, we must show that $B \notin \{A\}^+$.

finding keys

Given a table R and set of functional dependencies, to find the keys:

1. Consider every subset of attributes in R.
2. Derive the closure of each subset.
3. Find the superkeys which are the attributes whose closures contain every attribute in R.
4. Find the keys which are the minimal superkeys.

Some observations to eliminate some of the tedium:

- If a set of attributes is a key, then any superset cannot be a key (only a superkey).
- If an attribute is in *none* of the RHSs of the FDs, then it must be part of the key.

If an attribute is part of a key, then it is known as a **prime attribute**. Otherwise, it is a **non-prime attribute**.