CS561: Database Management Systems Notes

Steven DeFalco

Fall 2023

Contents

1 Introduction

Database management systems (DBMS) consist of **data**, **software** (programs such as data interfaces), and **environments** (operating systems). DBMS contains information about a particular enterprise. They are collections of interrelated data, a set of programs to access the data, and an environment that is both *convenient* and *efficient* to use. The *user* should only have to define what it is that they want from the database, whereas the *database* is responsible for defining how this query can be fulfilled; relational databases are good at this.

The three primary data models are ***entity-relationship model*** (diagrams), ***relational model*** (relational algebra), ***relational database*** (SQL).

Drawbacks to using **file systems** to store data include the following:

- data *redundancy* and *inconsistency* (multiple formats, duplication of information, etc.)
- difficulties in *accessing* data
- data isolation (multiple files and formats)
- concurrency issues (among multiple users)
- *integrity* problems
- atomicity of updates
- security problems (hard to provide varied levels of user access)

There are varying ***levels of abstraction*** in a database. The ***physical level*** defines how a record is stored. The ***logical level*** describes data stored in the database and the relationships among data. The ***view level*** is a way to hide details of data types and information for security purposes.

The ***schema*** is the logical structure of the database; this is analagous to type information of a variable in a program. ***Physical schema*** refer to database design at the physical level. ***Logical schema*** refer to database design at the logical level. An ***instance*** is the actual content of the database at a particular time; this is analagous to the value of a variable. ***Physical data independence*** is the ability to modify the physical schema without changing the logical schema.

Remark The schema of a table is the attributes of the table. For example, the schema of a table whose column titles are "A,B,C,D" is simply A,B,C,D.

Data manipulation languages (DML) are languages for accesing and manipulating the data organized in a DBMS. ***Procedural languages*** are ones in which the user specifies what data is required and how to get that data.

Declarative (nonprocedural) languages are ones in which the user specifies what data is required without specifying how to get such data. **SQL** is the most widely used query language.

A ***data definition language (DDL)*** is the specific notation for defining the database schema. The ***DDL compiler*** generates a set of tables stored in a data dictionary. Data dictionary contains metadata.

A ***relational database*** is based on the relational data model. Data and relationships among the data are represented by a collection of tables. These include both a **DML** and **DDL**. The most common relational database systems employ the **SQL** query language.

2 Entity-Relationship Model

A *database* can be modeled as a collection of entities or a relationship among entities. An ***entity*** is an object that exists and is distinguishable from other objects (e.g. specific person, company, even, plant). These *entities* have ***attributes*** (e.g. people hvae names and addresses). An ***entity set*** is a set of entities of the same type that share the same properties (e.g. set of all persons, companies). In the ER-model we refer to specific objects as *entities* which have *attributes* and are all a part of the entire *entity set*.

A ***relationship*** is an association among several entties. A ***relationship set*** is a mathematical relation among $n \geq 2$ entties, each taken from entity sets.

$$\{(e_1, e_2, \dots e_n) \,|\, e_1 \in E_1, e_2 \in E_2, \dots, e_n \in E_n\}$$

where $(e_1, e_2, \dots e_n)$ is a relationship.

2.1 Attributes

An ***attribute*** can also be property of a relationship set. For instance, the *depositer* relationship set between entity sets *customer* and *account* may have the attribute *access-date*. Relationshup sets that involve, two entity sets are ***binary*** (degree two). Relationship sets may involve more than two entity sets. Relationships between more than two entity sets are rare (i.e. most are binary).

An ***entity*** is represented by a set of attributes, that is descriptive properties possessed by all members of an entity set. ***Domain*** is the set of permitted values for each attribute. The **types of attributes** include the following:

- *simple* (atomic) and *composite* attributes
- *single-valued* and *multi-valued* attributes

- *derived* attributes (can be computed from other attributes)

When attributes are *simple* and *single-valued*, then we say that the data is in ***First Normal Form***.

2.2 Keys

A ***super key*** of an entity set is a set of one or more attributes whose values uniquely determine each entity. Once you have defined a *super key*, you can add attributes and it is still considered a *super key*. A ***candidate key*** of an entity set is a minimal super key (e.g. *cusomer_id* is a candidate key of *customer*). Candidate keys *only* contain the necessary attributes to make something unique. Although several candidate keys may exist, one of the candidate keys is selected to be the ***primary key***.

The combination of primary keys of the participating entity sets forms a super key of relationship set. This means a pair of entity sets can have at most one relationship for each access.

A relation schema may have an attribute that corresponds to the primary key of another relation. The attribute is called a ***foreign key***. Foreign key is the primary key of the parent table. Only values occurring in the primary key attribute of the **referenced relation** may occur in the foreign key attribute of the **referencing relation**.

2.3 E-R Diagrams

Rectangles represent entity sets. Diamonds represent relationship sets. Lines link attributes to entity sets and entity sets to relationship sets. Ellipses represent attributes: double ellipses represent multivalued attributes while dashed ellipses denote derived attributes. Underline indicates primary key attributes.

3 Relational Model

Formally, given sets $D_1, D_2, \ldots D_n$ a ***relation*** r is a subset of

$$D_1 \times D_2 \times \cdots \times D_n$$

Thus, a relation is a set of n-uples $(a_1, a_2, \ldots a_n)$ where each $a_i \in D_i$.

Each attribute of a relation has a name. The set of allowed valued for each attribtue is called the ***domain*** of the attribute. Attribute values are (normally) required to be ***atomic***; that is, indivisible. Domain is said to be atomic if all its members are atomic. The special value *null* is a member of every domain. The null value causes complications in the definition of many operations.

$A_1, A_2, \ldots A_n$ are attributes. $R = (A_1, A_2, \ldots, A_n)$ is a ***relation schema***. $r(R)$ denotes a relation r on the relation schema R.

The current values (***relation instance***) of a relation is specified by a table. An element t of r is a *tuple*, represented by a row in a table. Relations are unordered and thus the roder of tuples is irrelevant (tuples may be stored in an arbitrary order).

A ***database*** consists of multiple relations. Information about an enterprise is broken up into parts, with each relation storing one part of the information.
The two types of database management systems are OLTP (***Online Transactional Processing***) and OLAP (***Online Analytical Processing***). OLTP databases are update/change oriented (*most* common databases are this category): write oriented. OLAP databases are for viewing and analyzing the data: read oriented. In OLTP, a good table is a table about *one thing only*: ***third normal form***. For example, a good OLTP table will have a table containing information about customers only and customers only. OLTP tables are normalized, but OLAP tables must be denormalized and this can be achieved by using ***join operations***.

3.1 Query Languages

A ***query language*** is a language in which users request information from the database. ***Pure languages*** are relational algebra, tuple relational calculus, or domain relational calculus. *Pure langauges* form the underlying basis of query languages that poeple use.

3.2 Relational Algebra

Relational algebra is a procedural language with six basic operators:

- select: σ — **WHERE** in SQL
- project: $\prod$ — **SELECT** in SQL
- union: $\cup$ — **UNION** in SQL
- set difference: $-$ — **EXCEPT** in SQL
- cartesian product: $\times$ — **, (comma)** in SQL
- rename: ρ — **AS** in SQL

The operators take one or two relations as inputs and produce a new relation as a result.

Example 3.1 Translate the following relational algebra to SQL...

- $\sigma_{A=B \ \hat{} \ D>5}(r)$
  `SELECT * FROM r WHERE A=B and D>5`

- $\prod_{A,C}(r)$
  ```
  SELECT A,C FROM r
  ```
- $r \cup s$
  ```
  SELECT * FROM r
  UNION
  SELECT * FROM s
  ```
- $r - S$
  ```
  SELECT * FROM r
  EXCEPT
  SELECT * FROM s
  ```

When writing a ***SQL query*** always pull all your data togther into a single view. This means that you will start with a `FROM` clause. In OLTP, all of the tables are denormalized into a single table so that when making queries, you can have a simple `FROM` clause where the data is already in a single view. ***Join*** is a more selective version of ***cartesian product***. **Join** is essentially a cartesian product followed by a select operation. Both **join** and **cartesian product** are $\mathcal{O}(n^2)$, but **join** will very likely run faster in most cases.

Definition 3.1 (Rename Operation) ALlows us to name, and therefore to refer to, the results of relational algebra expressions. For example,

$$\rho_x(E)$$

returns the expressio E under the name X. If a relational algebra expression E has an arit n, then

$$\rho_{x(A_1,A_2,\ldots,A_N)}(E)$$

returns the result fo expression E under the name X, and with the attributes renamed to $A_1, A_2, \ldots, A_n$.

How to write a query (SQL / rel. algebra) based on English Description...

1. Identify *data elements*: attributes and columns (e.g. `cusomername`)
2. Identify the *sources* of the data elements in **Step 1** (e.g. `customer`)
3. Identify some *meaningful* relationships between/among the elements in **Step 2** (join/cartesian product followed by selection)

Example 3.2 (Query Translation Practice) Find all loans of over $ 1200.

$$\sigma_{\text{amount }>1200}(loan)$$

```
SELECT *
FROM loan
```

```
WHERE amt > 1200.
```

Find the loan number for each loan of an amount greater than $ 1200.

$$\prod_{\text{loan\_number}} (\sigma_{\text{amount}>1200}(loan))$$

```
SELECT loan_numb
FROM loan
WHERE amt > 1200.
```

Pushing down of selection operation is the idea where we perform selection as early as possible to attempt to make the table resulting from a join smaller. This can be used to help optimize SQL queries. Try to minimize the size of the table (reduce the number of rows and columns) that is generated with a join. Push down of the selection operation reduces the number of rows. Projection operation reduces the number of columns. ***Heuristic optimization*** is this process and is done by the DBMS.

Definition 3.2 (Set-Intersection Operation ($r \cap s$)) Set intersection is defined as $r \cap s = \{t \mid t \in r \text{ and } t \in s\}$. Where we assume that r,s have the same *arity* and attributes of r and s are compatible. Note that $r \cap s = r - (r - s)$.

Definition 3.3 (Natural-Join Operation ($r \bowtie s$)) Let r and s be relations on schemas R and S repsectively. Then $r \bowtie s$ is a relation on schema $R \cup S$ obtained as follows:

- Consider each pair of tuples t_r and r and t_s from s.
- If t_r and t_s have the same value on each of the attributes in $R \cap S$, add a tuple t to the result where
  - t has the same value as t_r on r
  - t has the same value as t_s on s

Example 3.3 (Natural-Join Example) Let $R = (A, B, C, D)$ and $S = (E, B, D)$.
Result schema $= (A, B, C, D, E)$
$r \bowtie s$ is defined as $\prod_{r.A,r.B,r.C,r.D,r.E}(\sigma_{r.B=s.B \wedge r.D=s.D}(r \times s))$

```
SELECT *
FROM r natural join s
```

or

```
SELECT y.A, y.B, y.C, y.D, S.E
FROM Y,S
WHERE Y.B = S.B
and Y.D = S.D
```

Definition 3.4 (Division Operation ($r \mid s$)) This operation is suited to queries that include the phrase "for all". Let r and s be relations on schemas R and S respectively where

- $R = (A_1, \ldots, A_m, B_1, \ldots B_n)$
- $S = (B_1, \ldots, B_n)$

The result of $r \mid s$ is a relation on schema $R - S = (A_1, \ldots, A_m)$

$$r \mid s = \{t \mid t \in \prod_{R-S} (r) \ \wedge \ \forall u \in s(tu \in r)\}$$

where tu means the concatenation of tuples t and u to produce a single tuple.

Remark Let $q = r \mid s$, then q is the largest relatin satisfying $q \times s \subseteq r$.

Definition 3.5 (Generalized Projection) Extends the prjection operation by allowing arithmetic functions to be used in the projection list

$$\prod_{F_1, F_2, \ldots, F_n} (E)$$

where E is any relational-algebra expression. Each of $F_1, F_2, \ldots F_n$ are arithmetic expressions involving constants and attributes in the schema of E.

An ***aggregation function*** takes a collection of values and returns a single value as a result. For example, average value, minimum value, maximum value, sum of values, and number of values are all aggregation functions.

Definition 3.6 (Aggregate Operation)

$$_{G_1, G_2, \ldots, G_n} g_{F_1(A_1), F_2(A_2, \ldots, F_n(A_n))}(E)$$

where E is any relational-algebra expression.

- $G_1, G_2, \ldots, G_n$ is a list of attributes on which to group (can be empty)
- Each F_i is an aggregate function
- Each A_i is an attribute name

Example 3.4 (Group By Examples)

- $g_{\text{sum}(c)}(r)$

```
SELECT sum(c)
FROM r
```

- $_{branch\_name} g_{\text{sum}(balance)}(account)$

```
SELECT branch_name, sum(balance)
FROM account
Group By branch_name
```

The result of an aggregation does not have a name, but we can use the rename operation to give it a name. For convenience, we permit renaming as a part of aggregate operation:

$$_{branch\_name}\mathcal{G}_{\text{sum}(balance) \text{ as } sum_balance}(account)$$

Definition 3.7 (Outer Join) an extension of the join operation that avoids loss of information. ***Outer Join*** computes the join and then adds tuples from one relation that does not match tuples in the other relation to the result of the join. This uses *null* values which signify that the value is unknown or does not exist. All comparisons involving *null* are false by definition.

Within this, there is left and right outer join. In ***left outer join***, the results include those from inner join and the entities from the first entity set that are not included in result of an inner join. Values that are missing (the reason they are not included in the inner join) will have null-values. There is also ***full outer join*** which is the union of both left and right inner join.

It is possible for tuples to have a null value, denoted by *null*, for some of their attributes. *null* signified an unknown value or that a value does not exist. The result of any arithmetic expression involving *null* is *null*. Aggregate functions simply ignore null values. For duplicate elimination and grouping, null is treated like any other value, and two nulls are assumed to be the same.

Comparisons with null values return the special truth value: ***unknown***. There are special rules involving the *unknown* truth value:

- OR

$$\begin{aligned}(unknown \text{ or } true) &= true,\\ (unknown \text{ or } false) &= unknown\\ (unknown \text{ or } unknown) &= unknown\end{aligned}$$

- AND

$$\begin{aligned}(true \text{ and } unknown) &= unknown,\\ (false \text{ and } unknown) &= false\\ (unknown \text{ and } unknown) &= unknown\end{aligned}$$

- (**not** *unknown*) = *unknown*

Definition 3.8 (Deletion) A delete request is expressed similarly to a query, except instead of displaying tuples to the user, the selected tuples are removed from the data base. Can only delete whole tuples; cannot delete values on only particular values. A deletion is expressed in relational algebra by:

$$r \leftarrow r - E$$

where r is a relation and E is a relational algebra query.

Definition 3.9 (Insertion) To insert data into a relation, we either:

- specify a tuple to be inserted
- write a query whose result is a set of tuples to be inserted

In relational algebra, an insertion is expressed by:

$$r \leftarrow r \cup E$$

where r is a relation and E is a relational algebra expression. The insertion of a single tuple is expressed by letting E be a constant relation containing one tuple.

Definition 3.10 (Updating) A mechanism to change a value in a tuple without charting all values in the tuple. Use the generalized projection operator to do this task

$$r \leftarrow \prod_{F_1, F_2, \ldots, F_l} (r)$$

Each F_i is either

- The l^{th} attribute of r, if the l^{th} attribute is not updated, or,
- if the attribute to be updated F_i is an expression, involving only constants and the attributes of r, which gives the new value for the attribute

4 SQL

An SQL relation si defined using the ***create table*** command:

$$\begin{aligned}\text{create table } r(&A_1 D_1, A_2, D_2, \ldots A_n D_n,\\ &(\text{integrity-constraint}_1),\\ &\ldots,\\ &(\text{integrity-constraint}_k))\end{aligned}$$

- r is the name of the relation
- each A_i is an attribute name in the schema of relation r
- D_i is the data type of values in the domain of attribute A_i

The ***drop table*** command deletes all information about the dropped relation from the database. The ***alter table*** command is used to add attributes to an existing relation

$$\text{alter table } r \text{ add } A\, D$$

where A is the name of the attribute to be added to relation r and D is the domain of A. All tuples in the relation are assigned *null* as the value for the

new attribute. The ***alter table*** command can also be used to drop attributes of a relation

$$\text{alter table } r \text{ drop } A$$

where A is the name of an attribute of relation r. Dropping of attributes is not supported by many databases.

SQL is based on a set and relational operations with certain modifications and enhancements. A typical SQL query has the form:

$$\begin{aligned}&\text{select } A_1, A_2, \dots A_n\\&\text{from } r_1, r_2, \dots, r_m\\&\text{where } P\end{aligned}$$

- A represents an attribute
- R_i represents a relation
- P is a predicate

This query is equivalent to the relation algebra expression

$$\prod_{A_1, A_2, \dots, A_n} (\sigma_p(r_1 \times r_2 \times \cdots \times r_m))$$

The result of an SQL query is a relation.

The ***select*** clause lists the attributes desired in the result of a query, and this corresponds to the projection operation of the relational algebra. SQL allows duplicaties in relations as wlel as in query results; to force the elimination of duplicates, insert he keyword **distinct** after select. The keyword **all** specifies that duplicates not be removed. An asterisk in the select clause denotes *all attributes*. The **select** clause can contain arithmetic expressions involvin the operation +,-,*, and /, and operating on constants or attributes of tuples.

The **where** clause specifies conditions that the result must satisfy; this corresponds to the selection predicate of the relational algebra. Compraison results can be combined using the logical connectives **and**, **or**, and **not**. Comparisons can be applied to results of arithmetic expressions. SQL includes a between comparison operator.

Example 4.1 (Between operator) Find the loan number of those loans with loan amounts bewtween $90,000 and $100,000.

```
1  select loan_number
2  from loan
3  where amount between 90000 and 100000
```

The ***from*** clause lists the relations involved in the query; this corresponds to the Cartesian product operation of the relational algebra.

SQL allows renaming relations and attributes using the **as** clause:

$$old - name \text{ as } new - name$$

SQL includes a string-matching oeprator for comparisons on character strings. The operator *like* uses patterns that are described using two special characters:

- percent (%). The % character matches any substring
- underscore (_). The _ character matches any character

SQL supports a variety of string operations such as

- concatentation (using "——")
- concerting from upper to lower case (and vice versa)
- finding string length, extracting substrings, etc.

We may specify ***desc*** for descending order or ***asc*** for ascending order, for each attribute; ascending order is the default

```
order by customer_name desc
```

The set operations **union**, **intersect**, and **except** operate on relations and correspond to the rleational algebra operations $\cup$, $\cap$, $-$. Each of the operations automatically eliminates duplicates; to retain all duplicates use the corresponding multiset versions **union all**, **intersect all**, and **except all**.

Remark Predicates in the **having** clause are applied after the formation of groups whereas predicates in the **where** clause are applied before forming groups.

`FROM` clause creates a single table by taking the cartesian product of all the specified table. `WHERE` clause creates a smaller table that is a subset of the table created with `FROM`. Then `GROUP BY` creates a new smaller table with the desired attributes. Finally, `HAVING` accesses the small table that is created by the `GROUP BY` clause. `HAVING` only has access to the output of `GROUP BY`.

SQL provides a mechanism for the nesting of subqueries. A ***subquery*** is a **select-from-where** expression that is nested within another query. A common use of subqueries is to perform tests for set membershup, set comparisons, and set cardinality.

Example 4.2 (Nested Subqueries) .

- Find all customers who have both an account and a loan at the bank

```
select distinct customer_name
  from borrower
  where customer_name in (select customer_name
                          from depositor)
```

Without using nested subqueries, you could use intersection or natural join such as seen here.

```
-- intersection
select customer_name from borrower
intersect
select customer_name from depositor


-- natural join
select distinct customer_name
from borrower natural join depositor
```

- Find all customers who have a loan at the bank who do not have an account at the bank

```
select distinct customer_name
  from borrower
  where customer_name not in (select customer_name
                              from depositor)
```

Definition 4.1 (Some Clause) F $\langle$comp$\rangle$ **some** $r \Leftrightarrow \exists t \in r$ such that (F $\langle$comp$\rangle$ t). Where $\langle$comp$\rangle$ can be: $<, \leq, >, =, \neq$.

Definition 4.2 (All Clause) F $\langle$comp$\rangle$ **all** $r \Leftrightarrow \forall t \in r$ (F $\langle$comp$\rangle$ t)

The **exists** construct returns the value **true** if the argument subquery is non-mepty.

$$\textbf{exists } r \Leftrightarrow r \neq \phi$$
$$\textbf{not exists } r \Leftrightarrow r = \phi$$

Remark Nested subqueries are computed once and the output is used as reference.

Correlated nested sub-queries expect a parameter from the outside query. When you can take a nested sub query out of the query and still run it, then we know that this is a ***non-correlated*** nested subquery.

Remark Succinct expressions lead to efficient evaluation.

The ***unique*** construct tests whether a subquery has any duplicate tuples in its result. SQL allows a subquery expression to be used in the **from** clause.

Definition 4.3 (With Clause) provides a way of defining a temporary view whose definition is available only to the query in which the with clause occurs.

In some cases, it is not desirable for all users to see the entire logical model. A ***view*** provides a mechanism to hide certain data from the view of certain users. Any relation that is not of the conceptual model but is made visible to a user as a *virtual relation* is called a view. A view is defined using the **create view** statement which has the form

create view v **as** ⟨query expression ⟩

where ⟨query expression⟩ is any legal SQL expression. The view name is represented by v. Once a view is defined, the view name can be used to refer to the virtual relation that the view generates. When a view is created, the query expression is stored in the database; the expression is substituted into queries using the view.

Remark When we create a view, a system table is created that maps the view name to the SQL code behind that view. It is when we access the view, that the SQL code is executed. This is to ensure ***data recency*** in the view.

One view may be used in the expression defining another view. A view relation v_1 is said to ***depend directly*** on a view relation v_2 if v_2 is used in the expression defining v_1. A view relation v_1 is said to ***depend on*** view relation v_2 if either v_1 depends directly to v_2 or there is a path of dependencies from v_1 to v_2. A view relation v is said to be ***recursive*** if it depends on itself.

Join operations take two relations and return as a result another relation. These additional operations are typically used as subquery expressiosn in the **from** clause. ***Join condition*** defines which tuples in the two relations match, and what attributes are present in the result of the join. ***Join type*** defines how tuples in each relation that do not match any tuple in the other relation are treated. *Join types* include inner join, left outer join, right outer join, and full outer jion. *Join conditions* include natural, on ⟨predicate⟩, and using$(A_1, A_2, \ldots, A_n)$.

5 Advanced SQL

Definition 5.1 (Built-in Data Types in SQL) Some of the following are built-in SQL data types:

- ***date***: Dates, containing a (4 digit) year, month, and day
- ***time***: Time of data, in hours, minutes, and seconds
- ***timestamp***: date plus time of day
- ***interval***: period of time

We can extract values from individual fields from date/time/timestamp using **extract**. Users can define types using the **create type** construct which creates user-defined type. **create domain** creates user-defined domain types.

Definition 5.2 (Domain constraints) the most elementary form of integrity constraint. They test values inserted in the database, and test queries to ensure that the comparisons make sense.

Large objects are stored as *large object*:

- **blob**: binary large object—object is a large collection of uninterpreted binary data
- **clob**: character large object—object is a large collection of character data

An ***assertion*** is a prediacte expressing a condition that we wish the database to always satisfy. When an assertion is made, the system tests it for validity, and tests it again on every update that may violate the assertion.

Some forms of ***authorization*** on parts of the database are:

- ***Read*** allows reading, but not modification of data
- ***Insert*** allows insertion of new data, but notmodification of existing data
- ***Update*** allows modification, but no deletion of data
- ***Delete*** allows deletion of data

6 Relational Database Design

A *good table* in OLTP should...

- Be about one *thing*
- Should have one primary key where all the other attributes serve to describe that primary key. Key depends on rest of the columns
- Table is in **BCNF** format or 3^{rd} normal form

Remark The primary key in a relationship table is made up of foreign keys.

Remark Two symptoms of a bad table are repetition or a lot of null values.

The process of splitting a bad table into smaller, more simple, tables is called ***decomposition***. If a decomposition is done well, then you will be able to natural join the new tables back together and get the original table back. Should a decomposition not meet these requirements, then it is called a ***Lossy Decomposition*** because there is a loss of integrity through this process. The goal is to perform a ***Loss Less Join Decomposition*** where a natural join of the new tables *will* return the original table.

In the case that a relation R is not in *good* form, we decompose it into a set of relations $\{R_1, R_2, \ldots, R_n\}$ such that

- each relation is in good form
- the decomposition is a lossless-join decomposition

The domain is ***atomic*** if its elements are considered to be indivisiable units. A relation schema R is in ***first normal form*** if the domains of all attributes of R are atomic. Non-atomic values complicate storage and encourage redundant (repeated) storage of data. Atomicity is actually a property of how the elements of the domain are used.

Functional dependencies require that the value for a certain set of attributes determines uniquely the value for another set of attributes. A functional dependency is a generalization of the notion of a *key*.

Definition 6.1 (Functional Dependencies) Let R be a relation schema

$$\alpha \subseteq R \text{ and } \beta \subseteq R$$

The functional dependency

$$\alpha \rightarrow \beta$$

holds on R if and only if for any legal relations $r(R)$, whenever any two tuples t_1 and t_2 of r agree on the attributes α, they also agree on the attributes β. That is,

$$t_1[\alpha] = t_2[\alpha] \Rightarrow t_1[\beta] = t_2[\beta]$$

We use functional dependencies to:

- Test relations to see if they are legal under a given set of functional dependencies: if a relation r is legal under a set F of functional dependencies, we say that r satisfies F.
- Specify constraints on the set of legal relations: we say that F holds on R if all legal relations on R satisfy the set of functional dependencies F.

A functional dependency is *trivial* if it is satisfied by all instances of a relation. In general, $\alpha \rightarrow \beta$ is trivial if $\beta \subseteq \alpha$. Given a set F of functional dependencies, there are certain other functional dependencies that are logically implied by F. For example, if $A \rightarrow B$ and $B \rightarrow C$, then we can infer that $A \rightarrow C$. The set of all functional dependencies logically implies by F is the ***closure*** of F. We denote the closure of F by F^+.

K is a ***superkey*** for relation schema R if and only if $K \rightarrow R$. K is a ***candidate key*** for R if and only if $K \rightarrow R$ and for no $\alpha \subseteq K, \alpha \rightarrow R$.

Definition 6.2 (Boyce-Codd Normal Form) A relation schema R is in BCNF with respect to a set F of functinal dependencies if for all functional dependencies in F^+ of the form

$$\alpha \rightarrow \beta$$

where $\alpha \subseteq R$ and $\beta \subseteq R$, at least once of the following holds:

- $\alpha \to \beta$ is trivial (i.e. $\beta \subseteq \alpha$)
- α is a superkey for R

Suppose we have a schema R and a non-trivial dependency $\alpha \to \beta$ causes a violation of BCNF. We decompose R into:

- $(\alpha \cup \beta)$
- $(R - (\beta - \alpha))$

If it is sufficient to test only those dependencies on each individual relation of a decomposition in order to ensure that *all* functional dependencies hold, then that decomposition is ***dependency preserving***. Because it is not always possible to achieve both BCNF and dependency preservation, we consider a weaker normal form, known as *third normal form.*

Definition 6.3 (Third Normal Form) A relation schema R is in third normal form (3NF) if for all:

$$\alpha \to \beta \text{ in } F^+$$

at least one of the following holds:

- $\alpha \to \beta$ is trivial (i.e. $\beta \in \alpha$)
- α is a superkey for R
- Each attribute A in $\beta - \alpha$ is contained in a candidate key for R

If a relation is in BCNF, then it is in 3NF. The third condition is a minimal relaxation of BCNF to ensure dependency preservation.

6.1 Functional Dependencies and Closures

Given a set F of functional dependencies, there are certain other functional dependencies that are logically implied by F. For example, if $A \to B$ and $B \to C$, then we can infer that $A \to C$. The set of all functional dependencies logically implied by F is the ***closure*** of F. We denote the closure of F by F^+. We can find all of F^+ by applying Armstrong's Axioms:

- If $\beta \subseteq \alpha$, then $\alpha \to \beta$ ***reflexivity***
- If $\alpha \to \beta$, then $\gamma\alpha \to \gamma\beta$ ***augmentation***
- If $\alpha \to \beta$, then $\beta \to \gamma$, then $\alpha \to \gamma$ ***transitivity***

These rules are ***sound*** (generate only functional dependencies that actually hold) and ***complete*** (generate all functional dependencies that hold).

We can further simplify manual computationa of F^+ by using the following additional rules.

- If $\alpha \to \beta$ holds and $\alpha \to \gamma$ holds, then $\alpha \to \beta\gamma$ holds ***union***
- If $\alpha \to \beta\gamma$ holds, then $\alpha \to \beta$ holds and $\alpha \to \gamma$ holds ***decomposition***
- If $\alpha \to \beta$ holds and $\gamma\beta \to \delta$ holds, then $\alpha\gamma \to \delta$ holds ***pseudotransitivity***

These rules can all by inferred from Armstrong's Axioms.

There are several uses of the attribute closure algorithm:

- To test if α is a superkey, we compute α^+ and check if α^+ contains all attributes of R.
- To check if a functional dependency $\alpha \to \beta$ holds, just check if $\beta \subseteq \alpha^+$. That is, we compute α^+ by using attribute closure, and then check if contains β.
- For each $\gamma \subseteq R$, we find the cosure γ^+, and for each $S \subseteq \gamma^+$, we output a functional dependency $\gamma \to S$

Definition 6.4 (Lossless-join Decomposition) A decomposition of R into R_1 and R_2 is lossless join if and only if at least one of the following dependencies is in F^+:

- $R_1 \cap R_2 \to R_1$
- $R_1 \cap R_2 \to R_2$

Given a set of attributes, α, the ***closure*** of α under F (denoted by α^+) as the set of attributes that are functionally determined by α under F. The algorithm to compute α^+, the closure of α under F is as follows:

```
1  result := α;
2  while (changes to result) do
3    for each β → γ in F do
4      begin
5        if B ⊆ result then result := result ∪ γ
6      end
```

Given some column (attribute) a, this algorithm will find all the columns that depend a. In other words, given the left side of some functional dependency, find all the attributes on the right hand side.

Remark To determine the superkeys of a schema given a schema, first run the attribute closure algorithm on every possible superkey (A, B, C, AB, AC, BC, ABC). Whichever closure returns the schema itself is a superkey. Essentially, the closure of the superkey must generate the entire schema. A candidate key will have no subsets whithin itself which also determine the schema