Contents

Chapter 1

Functional Dependency

1.1 Relation

Definition 1.1. A *relation* is an ordered pair (*S*,*R*), where S is an n-tuple of names of attributes, and R is a set of n-tuples with values for the attributes as described by S.

Given $T \in R$ and $S = (s_1, s_2, ...s_n)$, we denote the value for attribute s_1 in T as $T(s_1)$.

1.2 Functional Dependency

Definition 1.2. Given R and $S = (X, Y)$, we say that X determines Y, denoted $X \to Y$, if $T_1(X) = T_2(X)$ implies $T_1(Y) = T_2(Y)$ for any $T_1, T_2 \in R$, and we call this a *functional dependency*.

1.3 Armstrong's Axioms

1. *Reflexivity*: if $Y \subseteq X$ then $X \to Y$
2. *Augmentation*: if $X \to Y$ then $XZ \to YZ$ for any Z
3. *Transitivity*: if $X \to Y$ and $Y \to Z$ then $X \to Z$

1.4 Closure

Definition 1.3. The *closure* of a set F of functional dependencies, denoted $F+$, is the set of all functional dependencies that can be inferred from those in F.

1.5 Minimal Cover

Definition 1.4. The *minimal cover* G of a set F of functional dependencies, is the smallest set such that $G+ = F+$.

Chapter 2

Normal Forms

2.1 First Normal Form (1NF)

Definition 2.1. A *superkey* of a relation S, R is a set of attributes X such that $t_1(X) = t_2(X)$ if and only if $t_1 = t_2$. Such attributes are said to be *prime*.

A superkey is said to be *minimal* if it has the least number of attributes required to meet this condition. Such a minimal superkey is called a candidate key.

Definition 2.2. A set of relations is in *First Normal Form* if every relation has a minimal superkey.

2.2 Second Normal Form (2NF)

Definition 2.3. A *partial dependency* is a dependency of a non-prime attribute on a proper subset of a candidate key.

Definition 2.4. A set of relations is in *Second Normal Form* if it is in 1NF and it contains no partial dependencies.

2.3 Third Normal Form (3NF)

Definition 2.5. A *trivial dependency* is a dependency $X \to Y$ where $Y \subseteq X$.

Definition 2.6. A *transitive dependency* is a dependency inferred from the transitive axiom. If $X \to Y$ is a transitive dependency, we say Y is *transitively dependent* on X, otherwise Y is *directly dependent* on X.

Definition 2.7. A set of relations is in *Third Normal Form* if it is in 2NF and all functional dependencies $X \to Y$ are trivial, or X is a superkey, or all attributes $a \in (X - Y)$ are prime.

2.4 Boyce-Codd Normal Form (BCNF)

Definition 2.8. A set of relations is in *Boyce-Codd Normal Form* if it is in 2NF and all functional dependencies $X \to Y$ are trivial or X is a superkey.

Chapter 3

Relational Algebra

Given a relation (S, R), we define a formalism for retrieving information.

3.1 Select

The *selection* operator denoted σ takes two paramters: a set of attributes in S and a relation (S, R), and returns a new relation (S', R') in which S' contains only the specified attributes of S and R' contains modified tuples with only values corresponding to the attributes of S'. We denote this as $\sigma_{\text{X}}\text{Y}$ where X is the set of attributes to keep and Y is the relation.

3.2 Project

The *projection* operator denoted π takes two parameters: a set of conditions on the attributes of S and a relation (S, R), and returns a new relation (S, R') where $R' \subseteq R$ is the set of all tuples that meet the specified conditions. We donte this as $\pi_{\text{X}}\text{Y}$ where X is the set of conditions and Y is the relation.

3.3 Renaming Attributes

It is often useful to rename attributes and save intermediate relations. We do this with the $\leftarrow$ operator as in the following example.

Stuff(Noms,NotNoms) $\leftarrow \sigma_{\text{Edibles,Inedibles}}$ Items

This example selects the Edibles and Inedibles attributes of the Items relation, and saves them to a new relation Stuff and rename the attributes to Noms and NotNoms.

3.4 Cartesian Product

The *Cartesian Product* binary operator denoted $\times$ takes relations (S, R) and (S', R') and returns the relation $(S \cup S', R'')$ where R'' is a set of tuples $\{rr' | r \in R, r' \in R'\}$.

3.5 Natural Join

The *natural join* binary operator denoted $*$ takes relations (S, R) and (S', R') where $J = S \cap S' \neq \emptyset$, and returns $(S \cup S', R'')$ where R'' is the set of tuples $\{rr' | r \in R, r' \in R', r(J) = r'(J)\}$.

3.6 Join

The *join* operator denoted $\bigotimes$ takes relations (S, R), (S', R'), and a function f, and returns the relation $(S \cup S, R'')$ where R'' is the set of tuples $\{rr' | r \in R, r' \in R', f(r, r') = true\}$.

For example:

CALL $\bigotimes_{\text{CALL.portid = CALL\_FORWARD\_NUMBERS.portid}}$ CALL_FORWARD_NUMBERS

Joins CALL and CALL_FORWARD_NUMBERS when CALL.portid = CALL_FORWARD_NUMBERS.portid.

3.7 Aggregate Function

The *Aggregate Function* takes a set of attributes from the relation, a function list, and a relation and returns the result of applying the function to the tuples of the relation and grouping by the given attributes. Available functions are SUM, AVERAGE, MINIMUM, MAXIMUM, COUNT.

For example:

lineid $f_{\text{count scode}}$ SERVICE_SUBSCRIBERS

Returns the count of unique values for attribute scode, grouped by lineid on relation SERVICE_SUBSCRIBERS.

Chapter 4

Implementation

Databases store data persistently on a hard disk, and since most hard disks in use today are revolving magnetic disks, databases are highly optimized for this.

Revolving magnetic disks, as opposed to solid state disks, have a physical disk that spins under a moving read/write head. Such disks are organized into sectors (a slice of the disk), and each sector is further divided into blocks of a fixed number of bytes. In order to read to or write a block in a sector of the spinning disk, both the disk and head must move, which is the bottleneck in the speed of a drive.

Databases are optimized for this scenario, in that they minimize the number of blocks to be read from the drive, thus minimizing the amount of disk and head movement.

4.1 B+ Trees

The biggest optimization to minimize block retrievals is to store as much information within a block as possible. A B+ tree is a tree with a large branching factor wherein all data nodes are leafs.

In a B+ tree, a branching factor b is specified and largely determines the structure of the tree. The branching factor b is an upper bound on the number of children each internal node can have.

The data stored in a B+ tree must be associated with a fully ordered key, which determines the order of the data in the leaf nodes.

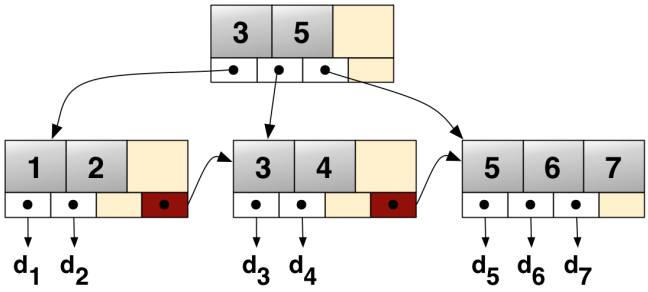


Figure 4.1: A B+ tree, image courtesy of Wikipedia

A B+ tree has many types of nodes:

- A leaf node which contains data.
- An internal root node which contains from 2 to b references to internal nodes. This node also stores the key values on which the internal nodes are split.
- An internal (non-root) node which contains between $\lceil b/2 \rceil$ references to either other internal or leaf nodes. If this node points to internal nodes, it contains the keys on which the internal nodes are split. If this node points to leaf nodes, it contains the keys of the leaf nodes it references.

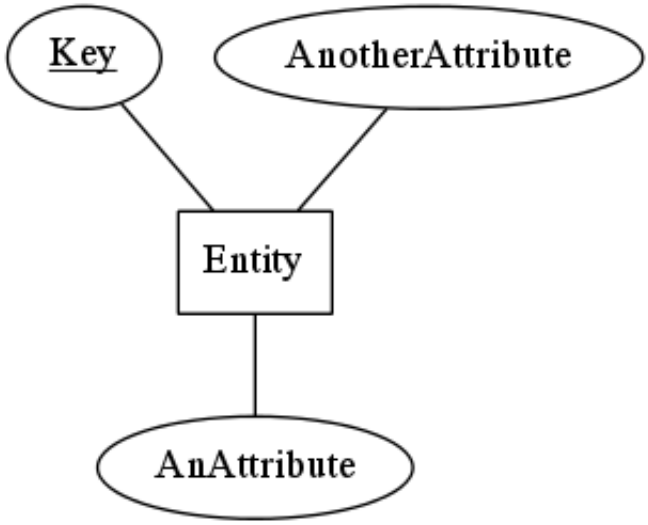
The smallest possible B+ tree contains just a single leaf node containing data.

In general, when a datum d is added to the tree, we find the location where d ought to be.

If that location is in the root leaf node, and the root leaf node contains less than b data we simply add it and we're done. If there are b other data on the root leaf node, it is split into 2 leaf nodes referenced to by an internal root node.

If d belongs in a non-root leaf node N, containing less than b other data we can simply add d and we're done. If N contains b other data, we must split N into two leaf nodes N_1 and N_2. We remove the reference to N from N's parent, and add the references to N_1 and N_2. Note that this may cause a cascade of node splits up to the root.

The complexity of insertion is $O(log_b(n))$. Note that our choice of b usually depends on the block size of the disk. We would like each node to occupy an entire block on disk, which minimizes block reads.

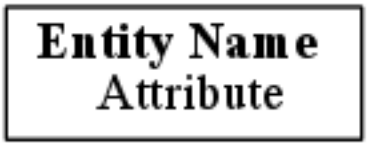


Appendix A

Entity Relational Diagrams

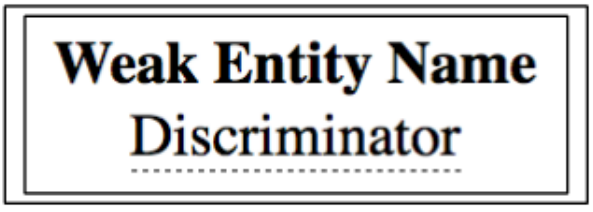
Entity Relational Diagrams are a visual diagramming language for describing entities using relational vocabulary.

To describe an entity, we draw a rectangle in which we write emphasized on the first line the name of the entity, and on the remaining lines the attributes of the entity.



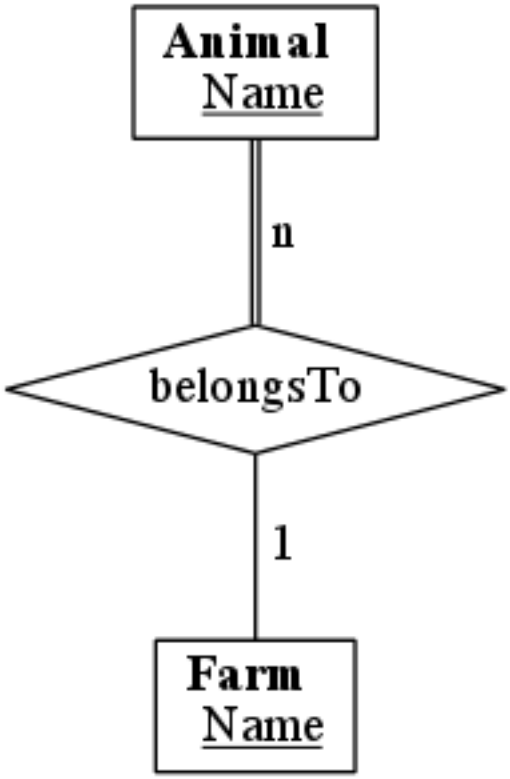
If it is a strong entity, we underline its natural key.

If it is a weak entity, we dashed underline its discriminator (or weak key), and draw a second rectangle around the first.



A variation is to write the attributes in ovals connected to the entity by lines.

To describe a relationship between two entities, we draw a rhombus in which we write the name of the relationship. We then draw lines between the rhombus and the entities it relates. We double the line between an entity and the relationship if the entity can't exist without being part of such a relationship. We further decorate the lines with cardinality: "1" meaning exactly one of this entity relates to one or many of the other entity in the relationship, "n" meaning many of this entity relates to one or many of the other entity.



We can also specify minima and maxima by "(min,max)". If the relationship itself has attributes, they will be drawn in ovals connected to the relationship by a line.

Appendix B

SQL

SQL is a Data Definition Language (DDL) and a Data Manipulation Language (DML), which means that SQL is suitable for both defining and manipulating large amounts of data.

B.1 SELECT

The `SELECT` operation in SQL is a combination of the *select* and *project* operations in relational algebra. The basic syntax is:

```
SELECT columns FROM table WHERE condition;
```

Where `columns` are columns in table `table`, and `condition` is a boolean condition like `col1 > 5`.

For example, given the following `coders` table schema:

name
codingAbility
birthDate

We could run the following query:

```
SELECT name, birthDate FROM coders
WHERE codingAbility > 9000;
```

Which would return all coders whose codingAbility is over 9000.

There are many extra options that can be specified on a `SELECT` query, the most useful of which is easily the `ORDER BY` option:

```
SELECT name, birthDate FROM coders
ORDER BY codingAbility DESC;
```

Which would return the names and birthDates of the coders in descending order of their codingAbility.

B.2 JOIN

A `JOIN` in SQL is just like a join in relational algebra. There are many joins in SQL, but by far the most common is the `INNER JOIN`, which only returns rows on which the join condition can be met.

Given the `coders` table described in the previous section, and the `coderAlias` table whose schema is as follows:

name
alias

We can make the following query:

```
SELECT alias, birthDate
FROM coders INNER JOIN coderAlias
ON coders.name = coderAlias.name;
```

Which would return the alias and birthDate of all coders who have a row in both the `coders` and `coderAlias` tables.

B.3 Aggregate Functions

Just like in relational algebra, SQL has the concept of aggregate functions such as: `MIN`, `MAX`, and `COUNT`.

A query to count the number of coders with each birthDate is as follows:

```
SELECT birthDate, COUNT(*) AS count FROM coders
GROUP BY birthDate;
```

The `GROUP BY` statement is crucial in this case, to aggregate the data by coder. Without the `GROUP BY` statement, the query would simply count how many birthDates were associated to each coder, which would be 1 in all cases because each coder has a unique row in the table.

Note that this query also renames the `COUNT(*)` column of the results as `count`, for convenience.