Database Systems Notes

1 Relational Algebra

Relation algebra can express the same statements as SQL `SELECT-FROM-WHERE` statements.

π	Projection
σ	Selection
$\times$	(*Cartesian/Cross*) Product
ρ	Renaming
$\bigcup$	Union
$\bigcap$	Intersection
$\setminus$	Difference
-	Difference
$\bowtie$	Natural Join

1.0.1 π: Projection

Projection is a *vertical* operation, allowing you to choose some *columns*.

Syntax:

$\pi_{\text{Set of Attributes}}(\text{relation})$

Any set of attributes not mentioned by the projection are discarded.

1.0.2 σ: Selection

Selection is a *horizontal* operation, allowing you to choose *rows that satisfy a condition*.

Syntax:

$\sigma_{\text{Condition}}(\text{Relation})$

This provides users with a *view* of data, hiding rows that do not satisfy the condition.

Consecutive selections are the same as a conjunction of the conditions in one selection; however, they can bring around different levels of performance. Consecutive selections are performed sequentially and eliminate rows that do not meet the criteria, whereas the conjunction means that the selection is performed once with a stricter condition.

1.0.3 $\times$: Cartesian Product

Where each row of two relations is *concatenated* to produce a new relation.

Syntax:

A Relation × Another Relation

A Cartesian product is the product of the two relations' sizes; meaning that Cartesian products can balloon in size.

1.0.4 ρ: Renaming

A useful tool that *aids* Cartesian products where the two relations have columns that go by the same name.

For example, a bank may have a table for all of a customer's accounts, and another table for all accounts open at some branch. These two tables could have a selection performed upon them to find all of a customer's accounts across all branches, but they both have an account name column. Renaming one allows for them to be distinguishable.

1.0.5 $\bowtie$: Natural Join

Joining two tables on *common attributes*.

This however can be expressed using a combination of projection, selection, renaming and the Cartesian product. It is more efficient to use a natural join however, similar to sequential selections.

2 SQL

All queries have the general format:

```
SELECT list, of, attributes
FROM list, of, relations
WHERE conditions
```

`WHERE` statements allow for tables to be **joined**.

Two relations can also be joined upon a condition, and are referred to as a *theta-join*. This is denoted as $R \underset{\theta}{\bowtie} S$. Theta-joins are essentially Cartesian-products with a condition subsequently applied to it, such that:

$$R \underset{\theta}{\bowtie} S \equiv \sigma(R \times S)$$

2.1 Table Management

2.1.1 Creating a Table

```
CREATE TABLE name (
     name type(size),
     .
     .
     .
);
```

2.1.2 Adding Entries to a Table

```
INSERT INTO name VALUES
    (val1, val2, val3);
```

2.1.3 Altering a Table's Attributes

```
ALTER TABLE name
    ADD COLUMN name type
    DROP COLUMN name;
```

2.1.4 Changing Values in a Table

```
UPDATE table
SET newValue
WHERE condition;
```

2.1.5 Removing Entries from a Table

```
DELETE FROM name
    WHERE condition;
```

2.1.6 Deleting a Table

```
DROP TABLE name;
```

2.2 Union Compatibility

Sometimes straightforward queries in English are a little more complicated than they are in relational algebra and in SQL.

2.3 Nested Queries

A nested query is also referred to as a *subquery* or a *inner* query.

This allows for a condition to be checked across both the main query, and the sub query.While this can be achieved by using joins, nested queries can be ideal for situations that have complex predicates.

2.4 Dependencies

The most common dependencies are *functional* and *inclusion* dependencies.

2.4.1 Functional Dependencies

Consider a relation R that has attributes X and Y. Y would be functionally dependent on X *iff* each value in X is associated with one value in Y. This is denoted $X \implies Y$.

Consider a database that holds *National Insurance* numbers and *employee names*. We would say that the *employee names* attribute is functionally dependent on the *National Insurance* numbers attribute. This is because the *National Insurance* number is *unique* to one person, whereas more than one person can have the same name.

So, say we have some set, U, containing all attributes of the relation R. The subset K of U is a *key* for R if satisfies the functional dependency $K \implies U$.

A *key* allows us to *uniquely identify* a tuple in a relation.

A key will always exists for any relation.

2.4.2 Inclusion Dependency

Beware; many **WORDS** appear in this section.

A key concept of this kind of dependency is *referential integrity*. This is when we expect that all values of some attribute in one table, all exist in some other table.

Referential integrity describes *inclusion dependencies*.

We say that an inclusion dependency holds when $R(A_1, \ldots, A_n) \subseteq S(B_1, \ldots, B_n)$; that if a tuple exists in R, then that exact tuple appears in S. These are referenced to as *foreign keys*, referencing $B_1, \ldots, B_n$ as the key for S.

In other words, a *foreign key* is a tuple that uniquely identifies a row in another table.

2.5 Keys

Intersects with the *Dependencies* section above.

2.5.1 Superkey

A column, or set of columns, that can be used to uniquely identify a row within a table.

2.5.2 Candidate Keys

A column or set of columns that is *minimal* and can be used to *uniquely identify* all records within a database.

2.5.3 Primary Keys

A *primary key* is a candidate key that *uniquely identify* all of a table's records. A primary *cannot be null* and must contain a *unique value for each row of data.*

This type of key is critical for relational databases to work as a concept. In SQL, the primary key can be defined in two ways.

After a variable declaration:

```
CREATE TABLE Student (
     name varchar(50),
     uun integer PRIMARY KEY
);
```

After a table declaration:

```
CREATE TABLE Movies (
     m_title varchar(30),
     m_director varchar(30),
     m_year smallint,
     m_genre varchar(30),
     PRIMARY KEY (m_title,m_year)
     );
```

The latter would be referred to as having a *compound primary key.*

A primary key does not allow for tuples to be inserted in to the table if they are the same as the primary key.

SQL allows for a key to be declared as `UNIQUE`, and is very similar to the `PRIMARY KEY` but `UNIQUE` still allows for null entries.

2.5.4 Foreign Keys

How *referential integrity* is enforced in SQL.

Say we have two tables created, we can declare a *foreign key* by stating that some attribute or attributes `REFERENCES` attributes in another table.

2.6 Joins

2.6.1 Inner Join

Similar to a logical `AND`, only records found in both tables will be returned.

2.6.2 Outer Join

The inverse to an inner join, an outer join will return records that are not in both tables. Similar to a logical `XOR`.

2.6.3 Left/Right Join

Returns all records that are in the left or right table, regardless if there is a record in the opposite table or not. If the is no equivalent equal record in the other table, a value of `NULL` shall be returned in that field instead.

2.6.4 Left/Right Outer Join

Returns all records that are in the left or right table, but does not match any record the opposite table.

2.6.5 Natural Join

Similar to the natural join found in relational algebra, where the join is based upon common columns in the two tables.

2.7 Aggregate Functions

Some examples of *aggregate functions* would be the `COUNT` and `SUM` functions. Care needs to be taken with aggregate functions as SQL tends to keep *duplicates* unless explicitly mentioned otherwise.

Aggregate functions can also affect the way that results are returned.

2.7.1 Order By

`ORDER BY` simply re-orders some output of an SQL query.

2.7.2 Group By

`GROUP BY` can sometimes seem to work in an underhanded way. Group by can point out to an aggregate function, on *what basis should attributes be aggregated.*

For example, say we have a table that details purchases made by customers on a website, and we want to find out how much each customer has spent.

The following SQL query would suffice:

```
SELECT name, SUM(amount)
    FROM sales
    GROUP BY name;
```

2.8 Cursor

A temporary portion of memory that used for executing a SQL statement. The set of rows that this statement returns is referred to as the cursor's *active set.*

An *implicit* cursor is used for `INSERT`, `UPDATE` and `DELETE` statements, as well as `SELECT` statements that return just one row.

An *explicit* cursor is used for `SELECT` statements that return more than one row.

3 Relational Calculus

Also known as *first-order predicate logic*. *Safe* relational calculus is equally as expressive as relational algebra.

Relational calculus consists of:

Relation Names	*Customers, Accounts*
Constants	*'London'*
Constraints	$\wedge, \vee, -$
Quantifiers	$\exists, \forall$
Bound Variables	$\exists x, \forall x$
Free Variables	*No quantifiers placed upon them*

When a query is without free variables, it is referred to as a *boolean query.*

A query is *safe* when it is known to give a finite answer across all databases.

The *active domain* of a table is the set of all its constants.

4 Translations

4.1 Relational Algebra to Relational Calculus

4.1.1 Base Relation

R becomes $R(x_1, \dots, x_n)$

4.1.2 Selection

$\sigma_\theta R$ becomes $R(x_1, \dots, x_n) \wedge \theta$.

4.1.3 Projection

Say we have a relation R with attributes x_1, x_2.

This becomes:

$$\exists x_2 \, R(x_1, x_2)$$

This is because attributes that are <u>not</u> projected, *become quantified.*

As we only want to project x_1, we quantify x_2, effectively elimanting it.

4.1.4 Cartesian Product

$R \times S$ becomes $R(x_1, \dots, x_n) \wedge S(y_1, \dots, y_n)$.

As all variables are *distinct*, output has only $m + n$ attributes.

4.1.5 Union

$R \cup S$, given that they are *union-compatible* becomes $R(x_1, \dots, x_n) \vee S(x_1, \dots, x_n)$.

4.1.6 Difference

$R - S$ becomes $R(x_1, \dots, x_n) \wedge \neg S(x_1, \dots, x_n)$.

5 Database Normalisation

Data redundancy is when the same piece of data is held in two or more separate places. Developers ideally want to be able to update all instances of some piece of data through one central access point, as it becomes an issue when one piece of data becomes inconsistent across several relations.

Database normalisation has the aim of minimising data redundancy, when some database schema is reduced into *normal form*.

5.1 1NF: First Normal Form

Criteria:

- Repeating groups in a relation are eliminated
- A separate table for each set of related data, identified by a primary key

5.2 2NF: Second Normal Form

Criteria:

- The table must be in *1NF*
- There should be no partial dependencies on any attributes of the primary key

This means that any of these attributes that are not a part of the primary key, but are directly related to that part of the primary key, should be placed in a different table.

For example, if we have a list that consists of `Manufacturer`, `Model` and `Manufacturer Country`; `Manufacturer Country` should be placed in a different table - because it is not a candidate key, but is directly dependent on `Manufacturer` - which would leave the relations in *2NF*.

5.3 3NF: Third Normal Form

Criteria:

- The table must be in *2NF*
- Every key that is not a part of a candidate key has a functional dependence on every *key* within the relation

5.4 BCNF: Boyce-Codd Normal Form

A slightly stronger version of *3NF*, where all redundancy based on functional dependencies has been removed.

For a relation to be in BCNF, every dependency $X \implies Y$:

- $X \implies Y$ is a *trivial functional dependency* $(Y \subseteq X)$
- X is the superkey of the relation

6 Serialisation

Given two schedules of transactions, that is read and write operations upon objects in a relation, we say that the two schedules are *conflict-serialisable* if they can be produce the same output if they can be run whilst overlapping, and if they are run sequentially without overlap.

When more than operation is placed upon an item in a relation, we break all the operations in to a list of pairs, such that the sequence `read(A), write(A), read(A), read(A)` would become:

```
(read(A), write(A)),
(write(A), read(A)),
(read(A), read(A)).
```

Conflicting pairs are `read-write`, `write-read` and `write-write`.

Conflict serialisability is enforced using *precedence graphs*, where each schedule has a node. An edge is drawn if an action in each schedule conflicts from the preceding action's node to the following action's node.

If the graph has no cycles, it is serialisable. Otherwise, it is not.

7 XML: Extensible Markup Language

XML is a format that is both *human* and *machine readable* and describes the structure of data.

XML doesn't define how data should be displayed, rather it provides a *model* for other applications to display it. This means that XML doesn't actually "*do*" anything, and depends on some other application to do something with it.

A similar language is HTML, which describes how data looks. XML focuses upon what data actually *is*.

XML allows for a developer to describe things, and ascribe *attributes* to them. This allows for a tree to be drawn representing the hierarchy.

7.1 DTD: Document Type Definition

A DTD is *grammar* that can be used in conjunction with XML documents, dictating which type of data some attribute can hold, what attributes an item has, and so on.