Part IA Databases Notes

1 Data models, Entity-Relationship (ER) model

1.1 DBMS

- **Primary service:** Persistent storage
- Provides an interface for the database
  - Database implementation can change without the application changing
  - An abstraction of resources or services used by applications
  - Stops application writers worrying about low level implementation
- Uses a query engine
  - Knows the details of the low level implementation
  - Optimises queries
- Typically implements CRUD operations and ACID transactions
- CRUD operations:

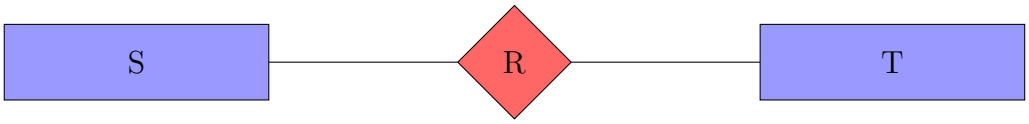
  Create: Insert new items into the database
  Read: Query the database
  Update: Modify objects in the database
  Delete: Remove data from the database
- **Redundant Data** is data that can be deleted and reconstructed from the remaining data, can cause the following anomalies
  - **Insertion Anomalies** - Occurs when certain attributes cannot be inserted into the database without the presence of other attributes
  - **Deletion Anomalies** - Exists when certain attributes are lost because of the deletion of other attributes
  - **Update Anomalies** - Occurs when one or more occurrences of repeated data is updated but not all not all occurrences.
- Fundamental trade-offs when designing a DBMS
  - Database engine must be optimised for a particular class of queries
  - A common trade-off: Query response vs update throughput
    - More redundant data can speed up queries
    - More redundant data causes update anomalies, update must search all of the database for all redundancies and update them.
  - Trade-offs change with time, as technology changes

1.2 ACID Transactions

Atomicity: Either all actions are performed or none are (even if the program crashes).
Consistency: Every transaction applied to a consistent database leaves it in a consistent state
Isolation: Transactions are protected from the effects of other concurrently executed transactions. As if the user did the transactions in sequence
Durablity: If a transaction completes successfully, then its effects persist

1.3 Entity-Relationship Model

- An implementation independent technique to describe the data stored in a database.
- The scope of the model is limited, it does not include all possible attributes
- **Entities** (square) represent the nouns of the model
- **Attributes** (oval) represent properties
  - Attributes can have their own attributes
- **Key** (underlined) an attribute that uniquely identifies an entity
  - Often automatically generated to be unique
  - May be algorithmically generated
  - A domain can have a natural key, beware safer to generate a new one
- **Relationships** (diamond) represent the verbs in the domain
  - Between two or more entities
  - Can have attributes
  - Ternary relationships can be represented by multiple binary relationships, by adding another entity
  - Relationships have cardinality



one-to-many	Every member of T is related to at most one member of S
many-to-one	Every member of S is related to at most one member of T
one-to-one	R is both many-to-one and one-to-many
many-to-many	No constraint

- **Weak entities** (double lined square) an entity whose existence depends on the existence of another entity
  - The relationship between a weak entity and the entity it depends on is called an **identifying relationship**

2 Relational Databases

2.1 Relational Algebra

- The Cartesian product is a set of 2-tuples:

$$S \times T = \{(s,\ t) \mid s \in S,\ t \in T\}$$

- A (binary) relation over $S \times R$ is any set R with

$$R \subseteq S \times R$$

- n-ary relations are used in databases If we have n sets (domains or attributes),

$$S_1,\ S_2, ...,\ S_n,$$

  then an n-ary relation R is a set

$$R \subseteq S_1 \times S_2 \times ... \times S_n = \{(s_1,\ s_2, ..., s_n) \mid s_i \in S_i\}$$

An n-ary relation can be thought of as a table, with each set S_i being an attribute.

- An attribute name A_i is associated with each domain S_i
- A record is a set of pairs of the attribute name and a value in the associated domain.

$$Record = \{(A_i,\ s_i) \mid i \in \mathbb{Z}^+,\ s_i \in S_i\}$$

- A database relation R is finite set of records

$$R \subseteq \{\{(A_1,\ s_1),\ (A_2,\ s_2),\ ...,\ (A_n,\ s_n)\} \mid s_i \in S_i\}$$

- R's schema is specified as $R(A_1 : S_1,\ A_2 : S_2\ ...\ A_n : S_n)$
- Relational Algebra is the following:

R	-	Base relation
$\sigma_p(Q)$	-	Selection, selects values where p holds
$\pi_X(Q)$	-	Projection, selects the attributes in the set X
$\rho_{\alpha \mapsto \alpha'(Q)}$	-	Renaming, renames α to α'
$R \cup S$	-	Union
$R \cap S$	-	Intersection
$R - S$	-	Difference, everything from R unless it is in S
$R \times S$	-	Product, all combinations of R_i and S_j. Not exactly Cartesian product because returns n-tuple not 2-tuple.
$R \bowtie S$	-	Natural join, set of all combinations of tuples in R and S that are equal on their common attribute names, see below.

- Relational algebra works on and returns sets
- A database query maps a collection of relation instances to a single relation instance
- Natural join
  - Equivalent to an AND
  - Can be defined in set notation, as follows
    Define:

$$(u.[A] = v.[A]) \quad \equiv \quad (u.A_1 = v.A_1) \wedge ... \wedge (u.A_n = v.A_n)$$

    That is that u and v are equal for each corresponding value in the column A, using this

$$R \bowtie S \equiv \{t \mid \exists u \in R,\ v \in S, (u.[B] = v.[B]) \wedge (t = u.[A] \cup u.[B] \cup v.[C])\}$$

  - In relational algebra

$$R \bowtie S = \pi_{A,B,C}(\sigma_{B=B'}(R \times \rho_{B \mapsto B'}(S)))$$

2.2 Keys

- A **superkey** is a set of attributes that uniquely identify a tuple. This can be expressed mathematically. Suppose $R(X)$ is a relational schema with $Z \subseteq X$. If for any records u and v in any instance of R

$$u.[Z] = v.[Z] \ \Rightarrow\ u.[X] = v.[X]$$

then Z is a superkey for R.

- A **key** is the smallest possible superkey
  - By definition no proper subset of a key is a superkey
- A relational schema can be written to reflect it's key, $R(\underline{Z},\ Y)$ shows Z is the key for $R(X \cup Y)$
- A **foreign key** is a set of attributes in a table that is a key for another table and for all instances of the tables all foreign keys match up with a tuple in other table.
  - This can be expressed mathematically, let $R(\underline{Z},\ Y)$ and $S(W)$ be relational schema with $Z \subseteq W$. Z is a foreign key of S in R if for any instance $\pi_Z(S) \subseteq \pi_Z(R)$.
- Schema of relationships

Relation R is	Schema
many-to-many	$R(\underline{X},\ \underline{Z},\ U)$
one-to-many	$R(\underline{X},\ Z,\ U)$
many-to-one	$R(X,\ \underline{Z},\ U)$
one-to-one	$R(\underline{X},\ Z,\ U)$ and/or $R(X,\ \underline{Z},\ U)$

- Multiple relationships can be implemented in one table by using a tag, constants denoting the relationship
- Using brute for a join has $O(n^2)$ complexity, instead indices are used
  - An index is a data structure that greatly reduces the time needed to locate records
  - Index speeds up reads, but will slow down writes

2.3 SQL

- SQL is based on multisets not sets
- Searching

```
SELECT <field names>
FROM <tables>
WHERE <conditions>;
```

- Ordering search results

```
SELECT <field names>
FROM <tables>
WHERE <conditions>
ORDER BY <field names> ASC/DESC;
```

- Inner join

```
SELECT <field names>
FROM <table1>
INNER JOIN <table2>
ON table1.columnname = table2.columnname
WHERE <conditions>;
```

- String match

```
SELECT <field names>
FROM <tables>
WHERE <attribute> like '%<STRING>%';
```

- Limit number of results

```
LIMIT <NUMBER>;
```

- Count number of items in a group (counts all if no groups)

```
COUNT(*);
```

- Self joins

```
SELECT G1.genre AS genre1, G2.genre AS genre2, count(*) AS total
FROM genres AS G1
JOIN genres AS G2 ON G1.movie_id = G2.movie_id
WHERE G1.genre <> G2.genre
GROUP BY genre1, genre2
ORDER BY total desc
LIMIT 10;
```

- Sub query Use the IN statement

```
SELECT title
FROM movies
WHERE id NOT IN (
    SELECT movie_id
    FROM languages
    WHERE language = 'English' );
```

- An expression with NULL in it will always evaluate to NULL. Instead use IS,

```
...
position IS NULL
```

- Left and right joins
  - For tables T1 and T2 the query T1 LEFT JOIN T2 fills in the missing T2 columns with NULL values, while T1 RIGHT JOIN T2 fills in the missing T1 columns with NULL values
- Full outer join
  - T1 FULL OUTER JOIN T2 fills in the missing columns with NULL values on both sides.
- NULL is a placeholder
  - NULL is not a member of any domain (type)
  - Use three value logic, NULL is treated as don't know
  - The possible interpretations of NULL
    - There is a value, but we don't know it
    - No value is applicable
    - The value is known, but you aren't allowed to see it

3 Graphs

3.1 Mathematical Relations

- Composition, give two binary relations

$$R \subseteq S \times T$$
$$\mathrm{Q} \subseteq T \times U$$

we can define their composition $Q \circ R \subseteq S \times U$ as

$$Q \circ R \equiv \{(s,\ u) \mid \exists t \in T,\ ((s,t) \in R) \wedge ((t,u) \in Q)\}$$

- A (partial) function $f \in S \to T$ can be thought of a binary relation where $(s,\ t) \in f$ iff $t = f(s)$
  - That is f is a function that maps S to T
- Suppose R is a relation where if $(s,\ t_1) \in R$ and $(s,\ t_2) \in R$, then it follows that $t_1 = t_2$. In this case R represents a function
- Given functions $f \in S \to T$ and $g \in T \to U$ their composition $g \circ f \in S \to U$ is defined by $(g \circ f)(s) = g(f(s))$
- We can write $Q \circ R$ as $R \bowtie_{2=1} Q$ therefore joins are a generalisation of function composition
- In a binary relation $R \subseteq S \times S$, we can define iterated composition as,

$$\begin{aligned} R^1 &\equiv R \\ R^{n+1} &\equiv R \circ R^n \end{aligned}$$

3.2 Directed Graphs

- $G = (V,\ A)$ is a directed graph
  - V is a finite set of vertices (or nodes)
  - A is a binary relation over V, that is $A \subseteq V \times V$
  - If $(u,\ v) \in A$ then there is an arc between u and v
  - The arc $(u,\ v) \in A$ is also known as a directed edge or relationship of u to v
- The relationship A can be composed, this is equivalent to following two consecutive edges
- A sequence of consecutive edges is often written in the form $v_1 \to v_2 \to ... \ v_{k+1}$
- If $(u,\ v) \in A^k$ then there is at least one path of length k between u and v
- Informally transitive closure gives you the set of all places you can get to from a starting position
  - Denoted as R^+
  - A relation is transitive if,

$$(x,\ y) \in R^+ \wedge (y,\ z) \in R^+ \to (x,\ z) \in R^+$$

  - Formally, it is the smallest binary relation on S such that $R \subseteq R^+$ and R^+ is transitive
  - The transitive closure is defined as,

$$R^+ = \bigcup_{n \in \{1,\ 2, ...\}} R^n,$$

    that is,

$$R^+ = R^1 \cup R^2 \cup ... \ \cup R^k$$

  - transitive closure can't be computed in Relational Algebra or SQL (without recursion)

4 Other Database Systems

4.1 Reasons for Read Optimised Systems

- When you might want a read optimised database
  - Data is seldom updated, but often read
  - Your reads can be slightly out of sync with the write orientated database, and periodically extract the read-orientated data from the write-orientated one
  - Used in a FIDO (fetch intensive data organisation)

4.2 Document Orientated Databases

- A document-orientated (aggregate-orientated) database stores data in the form of semi-structured objects
  - If all data were stored in a single table then attempting access by a key would require multiple look ups
  - Instead all data pertaining to a key is stored in a semi-structured object
- If there is value in the links between your documents (eg. in a social network) then don't use a document based system
- Document-orientated systems often use JSON

object	{}
	{ members }
members	pair
	pair , members
pair	string : value
array	[]
	[elements]
elements	value
	value , elements
value	string
	number
	object
	array
	true
	false
	null

- A key-value store simply maps a key to a block of bytes
  - The system does not interpret the bytes, this is left up to the application
  - Some do however interrupt the data for example indexing it

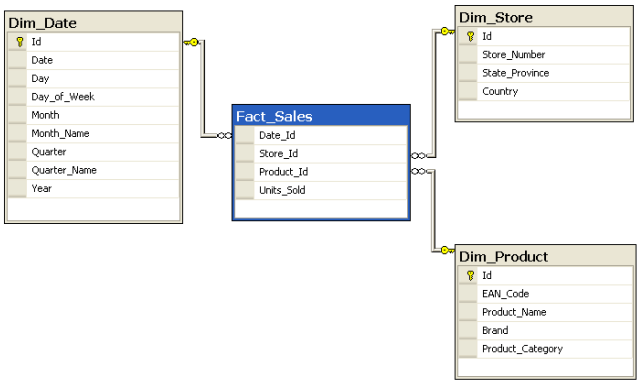
4.3 OLAP & OLTP

- **OLTP** - Online Transaction Processing
  - Optimised for updates
- **OLAP** - Online Analytical Processing
  - Commonly associated with decision support and data warehousing
- Differences between OLAP and OLTP

	OLAP	OLTP
Supports	analysis	day-to-day operations
Data is	historical	current
Transactions mostly	reads	updates
optimised for	reads	updates
data redundancy	high	low
database size	humongous	large

4.4 Aggregates

- SQL doesn't handle pivot tables (like spread sheets) well
  - Flat tables (relations) are great for processing, but not human readable
  - Pivot tables (cross tabulations) are hard to process but human readable
  - In pivot tables some table values become column and row names
- A data cube is instead used
  - Data is modelled as an n-dimensional hypercube
  - Each dimension is associated with a hierarchy
    - For example maybe location, a tree getting more specific further from the root
    - Drill-down to the more specific data
    - rollup/ aggregate to the more general (global) data
  - Each point records facts - can be lots of data
  - Aggregation and cross-tabulation possible along all dimensions
- Star Schema is used to implement data warehouses



  - Has a central fact table in which records have keys for dimension tables and measurements (atomic data)
  - Dimension tables have further data and are referenced by the fact table
  - In practice fact tables are usually very large
- Column-oriented implementations
  - In column-oriented implementations the system only access one column at a time
  - **Row-store** - Easy add/modify data but reads unnecessary data
  - **Column-store** - Only reads relevant data but tuple writes require multiple accesses
  - Best for read-mostly, read-intensive, large data repositories

4.5 Distributed databases

- Advantage of distributed data
  - **Scalability** - Data or workload can be too large for one machine
  - **Fault tolerance** - Can survive failure of some machines
  - **Lower latency** - Data can be located closer to the user

- Two ways of achieving distributed database (often a combination of both),
  - **Replication** - The data is duplicated
  - **Partitioning** - The data is split
    - Good for read-orientated databases
    - Partitions are often replicated

  It is hard for a highly distributed database to achieve all of the CAP concepts
  - **Consistency** - All reads return data that is up-to-date
  - **Availability** - All clients can find some replica of the data
  - **Partition tolerance** - The system continues to operate dispute network failure or failure of parts of the system
- Assume that network partitions and other connectivity problems will occur
- Implementing transactional semantics is very slow and difficult
- Trade-off between availability and consistency
- **Eventual Consistency** - If update activity stops the system will eventually reach a consistent state.