Databases

Storing Data

Format	Where
Fixed field record	Used in punch cards, each record can store 80 characters, some of the 80 characters are allocated to different fields.
Comma separted value record	More flexible than fixed field records, as there is no character limit to the value of a field.

A **simple associative store** will have the **API formal specification**.

```
store: string * string -> unit
retrieve: string -> string option (* either the value stored, or nothing at all *)
```

Definitions

Keyword	Meaning
Value	What is being stored.
Field	A place to hold a value.
Schema	A strongly typed specification of the database.
Key	The field(s) used to locate a record.
Index	A derived structure used to quickly find relevant record.
Query	A lookup function.
Update	A modification of data.
Transaction	An atomic change of a set of field.

An update is often implemented as a transaction.

Abstraction and Interfaces

We hope to have a fairly narrow interface that can map onto many operations. In an ideal world:
- All operations are done through the interface.
- The interface never change.

But in real world, large database projects are often a mess, and fixing the mess sometimes requires changing the interface. Changing interfaces break applications!

Logical Arrangement of a Database

The **DBMS** uses the **disc drive**, which is an abstraction over the **secondary storage**. This gives a consistent, nonchanging API which applications can use.

This API is often in form of SQL queries, SQL injection is an example of insecurity in the interface.

Operating System View of a Database

The **DBMS** is a process running in userspace.
- An **application** communicates with the **DBMS** through **kernel space communication**.
- Then the **DBMS** use **kernel space drivers** to access the **DBMS**.

An alternative implementation have the **DBMS** access a non-OS partition directly.

Data can be stored in primary store, secondary store, or distributed.

Definition
Big data is data too big to fit in primary store.

DBMS Operations

Operation	Description
Create	Insert new data
Read	Query the database
Update	Modify objects in the database
Delete	Remove data

Management Operations

- Create/change schema
- Create views
- Indexing/stats generation
- Reorganise data layout and backups

Amount of Writing

Database implementation choice depends on the amount of writing.

- A database can have a lot of lookups but rarely changes (e.g. library catalogue).
- **Transaction optimised**: concurrent queries and updates, they will affect the consistency of reads.

Definition
Atomicity is where changes are apparent to all users at once. An overhead is needed to achieve this.

- **Append only journal**: inverse of a transaction optimised DB, data is never updated.

Consistency Checks

Consistency rules includes value range check, foreign key referential integrity, value atomicity (all values are atomic), entity integrity (no missing fields).

Types of Databases

Relational Database

Consists of 2D tables.

- One row per record (a.k.a. a tuple), each with a number of fields.
- A table can have a schema.
- The ordering of the fields are unimportant.

Distributed Databases

Database can be spread over multiple machines.

Benefit	Description
Scalability	Dataset may be too large for a single machine.
Fault tolerance	Service can survive the failure of some machines.
Lower latency	Data can be located closer to the users.

Downsides: overhead after an update to provide a consistent view.

Keyword	Meaning
Consistency	All reads return data that is up to date.
Accessibility	All clients can find some replica of the data.
Partition tolerance	The system continues to operate despite arbitrary message loss or failure of parts of the system.

It is impossible to achieve all 3 in a distributed database. *(why?)*

System that offers **eventual consistency** will eventually reach a consistent state once activity ceases.

ER Model

ER model is an implementation independent technique of describing the data we store in a database.

Definitions

Keyword	Meaning	Symbol
Entity	Model things in the real world	Rectangle
Attributes	Represent properties	Oval
Key	Uniquely identifies an entity instance.	Underline

A key can be **composite**. Examples of **natural keys** are a person's name and nation ID unumber. They may not be unique, it is often better to use a **synthetic key** - beware of using keys that are out of your control.

Definition
A **synthetic key** is auto-generated by the database and only has meaning within the database.

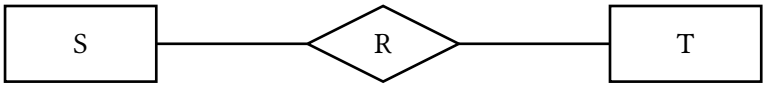
The **scope** of the model is the limited subset of all the real world attributes of the object.

Relationship Cardinalities

Many-to-many Relationships

- Represented in undecorated lines.
- Any S can be related to 0 or more T.
- Any T can be related to 0 or more S.

Note relations can also have attributes.



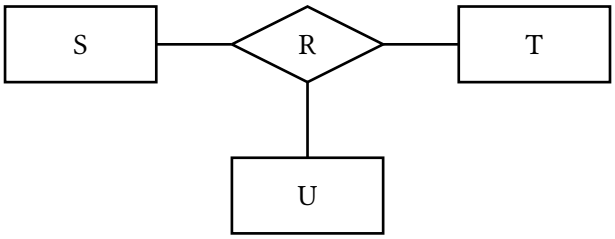
Rules for Modelling

- An attributes exists at most once for any entity or relation.
- Rule of atomicity: every value in a box must be atomic (a.k.k. 1NF)

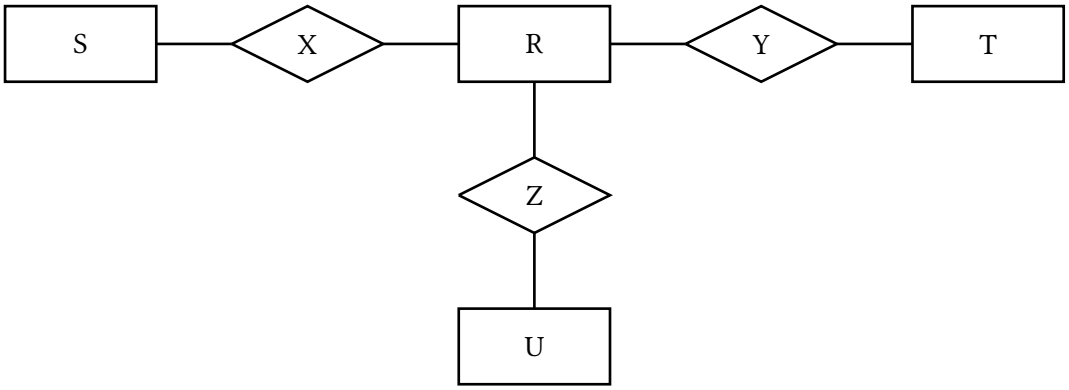
Do not put a comma separated list in the value, more likely than not we will need to break up the list again when searching, which requires text processing.

Tenary Relationships

They may be appropriate, depending on use.



What about 3 binary relationships?



Which one is appropriate depends on the situation, each entity should represent a real world object.

- **Many-to-one** relationships are denoted by an arrow towards the one end.
- **One-to-one** relationships are denoted by two arrow, one at each end.

If R is both many-to-one and one-to-many between S and T, it is one-to-one between S and T.

Definition
A **bijection** is where every member in one set is related to one member of the other set.

One-to-one cardinality doesn't mean one-to-one correspondence: not all elements in S are related to an element in T.

Weak Entity

Definition
The **weak entity** cannot exist without the **strong entity** existing.

- E.g. an alternative title (weak) for a movie (strong).
- Has **discriminators**, not keys. Combined with the **key** from the strong entity uniquely defines the weak entity.

Entity Hierarchy (OO-like)

- Subentities inherit the attribues and relations of the parent entity.
- Multiple inheritance is possible.
- We represent the relation with an *IS A* relation (upside down triangle in diagram).

The Relational Model

Before the relational model, you need to know about the data's low level representation to write a database application.

- We give the user a model of data and language for manipulating the data that is independent of the implementation details.
- The model is based on mathematical relations.

Mathematical Relations

Definitions

- The **cartesian product** is the set of all possible pairs.

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

- A **binary relation** of $S \times T$ is any set R with

$$R \subseteq S \times T$$

- R is a **binary relation**.
- S and T are referred to as **domains**.

We are interested in finite relations R that are explicitly stored in a database, which *does not include* infinite relations like the modulus. (e.g. $6 \bmod 5 = 1$)

N-ary Relations

If we have n domains, then n-ary relation R is the set

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

Definition

Tabular presentation of a relation is where each tuple has a column number.

1	**2**	**...**	$\boldsymbol{n}$
a_1	b_1	...	x_1
a_2	b_2	...	x_2
a_3	b_3	...	x_3
a_4	b_4	...	x_4

Named Columns

Because referring to tuples by column number is annoying.

- Associate an attribute name A_i with each domain in S_i.
- Use records instead of tuples.

Definition

A **database relation** is the finite set

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

We specify R's schema as $R(A_1 : S_1, A_2 : S_2, ...A_n : S_n)$.

Example: Students

Using the schema *Students(name: string, sid: string, age: integer)*:

```
Students = {
  { (name, "sirius"), (sid: "12345"), (age: 19) }
}
```

Query Language

- The input of a query language is a collection of relatin instances $R_1, R_2, ..., R_k$.
- The output is a single relational instance $Q(R_1, R_2, ..., R_k)$.

We want to create a high level query language Q that is independent of the data, there are many ways to do that.

The Relational Algebra Abstract Syntax

It is generally a tree structure

$Q ::=$

$\mid R$	base relation
$\mid \sigma_{p(Q)}$	selection
$\mid \pi_{x(Q)}$	projection
$\mid Q \times Q$	product
$\mid Q - Q$	difference
$\mid Q \cup Q$	union
$\mid Q \cap Q$	intersection
$\mid \rho_{M(Q)}$	rename

- $p : x \mapsto$ bool is a boolaen predicate.
- $X = \{A_1, A_2, ...\}$ is a vector (a set of attributes).
- $M = \{A_1 \mapsto B_1, A_2 \mapsto B_2, ...\}$ is a list of mappings that takes in one attribute name, and output another attribute name.

Definition
Q is **well formed** if all attribute names of the result are distinct.

- $Q_1 \times Q_2$ requires the subqueries to share no column names.
- $Q_1 \cup Q_2$ requires the subqueries to share all column names.

Mapping RA to SQL

RA is based directly on the underlying set theory.

- Formal semantics/specification works by mapping the query language back to set theory.

Operators

RA	SQL	Notes

$\sigma_{A>12}(R)$	`SELECT DISTINCT *`
	`FROM R`
	`WHERE R.A > 12`
		-

$\pi_{B,C}(R)$	`SELECT DISTINCT B, C` `FROM R`	-
$\rho_{\{B \mapsto E, D \mapsto F\}}(R)$	`SELECT A, B as E, C, D as F` `FROM R`	If we want to swap 2 rows, we will need to rename them one at a time.
$R \cup S$	`(SELECT * FROM R) UNION` `(SELECT * FROM S)`	R and S must have the same schema.
$R \cap S$	`(SELECT * FROM R) INTERSECT` `(SELECT * FROM S)`	
$R - S$	`(SELECT * FROM R) EXCEPT` `(SELECT * FROM S)`	In set theory, $R - S = R \cap (-S)$, but this doesn't make sense on databases, as $-S$ would mean everything that is not in S.

$R \times S$	`SELECT A, B, C, D`
	`FROM R`
	`CROSS JOIN S`
		In set theory, the cartesian product is a set of tuples. In SQL, that tuple is flattened.

Note
The **`DISTINCT`** keyword removes duplicates: each A, B, C, D might be unique, but B, C may be not. Not being distinct is fine in SQL but not fine in set theory as a set contains no duplicates.

The Natural Join

If we ignore domain types and write a relation schema as $R(A)$, where $A = \{A_1, A_2, ...\}$.

- $R(A, B) = R(A \cup B)$ assumes $A \cap B = \emptyset$
- $u.[A] = v.[A]$ means $u.A_1 = v.A_1 \wedge u.A_2 = v.A_2 \wedge ....$
  - For the fields specified, the two records completely agree with each other.

For $R(A, B)$ and $S(B, C)$, define the natural join $R \bowtie S$.

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

```
SELECT *
FROM R
NATURAL JOIN S
```

- If `NULL` values exists, then equality gets more complicated (how?), there are further variations of natural joins (left/right/inner/outer).
- If we want to use a custom predicate, use `WHERE` instead of `NATURAL`.

Relational Data Model

How do we store data in a table?

Data Redundancy

One approach is to store everything in one big table.

A big table is a redundant data representation, and redundancy can lead to inconsistencies.
(Example: movies database)

- We should be able to insert a person without knowing his role in a film.
- If we delete all films about a director, we lose information about the director.
- If a director's name is mispelled, we need to update it for all films.

Rule
Rule for removing **data redundancy**: store one fact in one place.

It also cause performance issues:

- To maintain consistency for every user, a simple transaction involves many things to do.
- Many records have to be locked to maintain atomicity, so it is less concurrent than it could be.

Breaking down tables reduces redundancy, but we will need to do joins which is expensive.

- The naive implementation requires doing a cartesian product, which has quadratic time.
- Actual database implementations uses heuristics to reduce time complexity.

Rule
The record should semantically depend on the key.

E.g. timestamp of birth should not be used as the key for a person, as a person's name does not depend on the timestamp.

An **all key table** is needed to store a many-to-many relation.

Relational Keys

Definition
A **relational key** is a unique handle on a record.

Let $R(X)$ is a relational schema and $Z \subseteq X$. If for any record u and v:

$$u[Z] = v[Z] \implies u[X] = v[X]$$

Then Z is a **superkey** of R.

If no proper subset of Z is a key for R, then Z is a **key** for R. Then for $R(Z \cup Y)$, we write

$$R(\underline{Z}, Y)$$

Foreign Key

Suppose we have $R(Z, Y)$, let $S(W)$ be a relational schema where $Z \subseteq W$. We say Z represents a foreign key in S for R if for any instance

$$\pi_Z(S) \subseteq \pi_Z(R)$$

Think of the foreign key as a pointer.

A foreign key is denoted with an overline.

Definition
A database has **referential integrity** when all foreign key constrains are satisfied.

Simplifying Relations

We can combine tables together using **type tags** to reduce the number of tables.

Before	After
$R\big(\underline{X, Z}, U\big)$	$RQ\big(\underline{X, Z, \text{type}}, U, V\big)$
$Q\big(\underline{X, Z}, V\big)$	Where $\text{type} = \{u, v\}$.

Notice all attributes are still semantically dependent on the key.

- Advantage: one less table, so less joins needed.
- Disadvantage: if a record does not have a relation, then the field storing the attribute will be `NULL`.

Transactions

Definition
A **transaction** on a DB is a series of queries or changes that externally appears to be atomic.

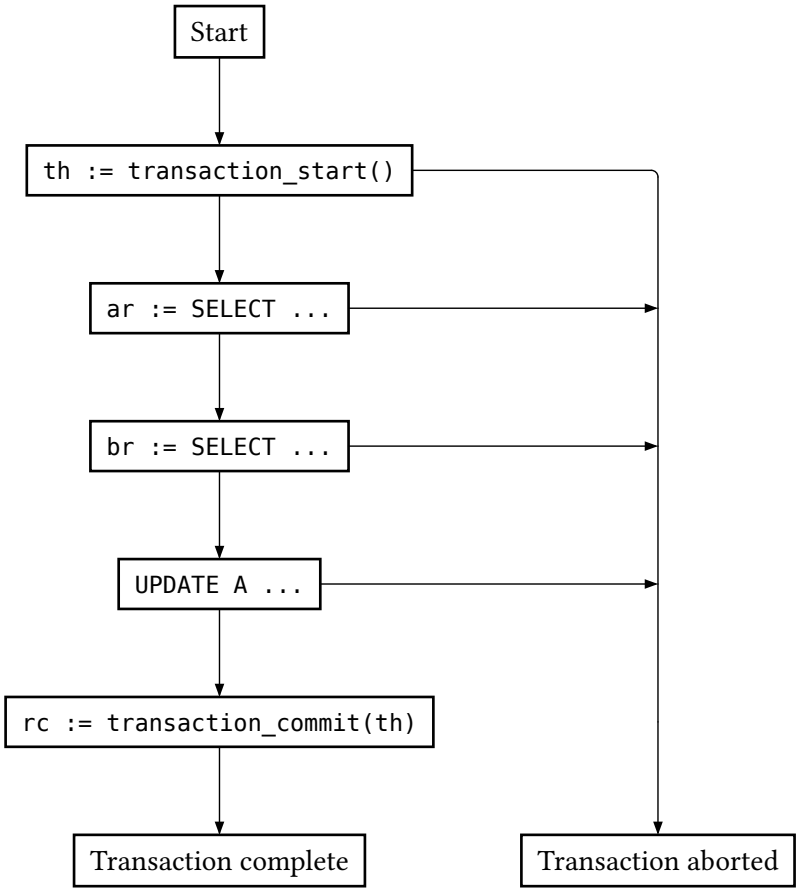
External transactions (i.e. side effects such as sending SMS messages) are not atomic.

Aborting

Some DBs allows aborting a transaction before committing.

This ensures all actions in the commit are undone and invisible externally.

In **optimisic locking** (a type of non-locking concurrency), transaction can abort and the client will be forced to restart the transaction.



ACID

Definitions

Keyword	Definition
Atomicity	All changes are performed as if they are a single operation.
Consistency	Every transaction leaves the DB in a consistent state.
Isolation	Intermediate state is invisible to other transactions, transactions appear **serialised**.
Durability	Once a transaction is complete, changes to data persists even in a system failure.

BASE

NoSQL DBs often have a weakened form of ACID.

Definitions

Keyword	Definition
Basically Available	Availability over consistency.
Soft state	Reads may give different values as data may change at any time (as opposed to only at the end of a transaction.)
Eventual consistency	If everyone stops updating it, everyone will see the same view.

So it is not very good when a lot of people are writing to the DB simultaneously.

Locks

Definition
A **lock** is a special type of hardware/software primitive that provides *mutual exclusion.*

- Resource are locked for exclusive access by one transaction and released after use.
- Other contendents will have to wait, delaying their completion. This **degrades throughput**.

Redundancy

Definition
If a data can be deleted then reconstructed by data in other parts of the DB, then it is **redundant**.

- If there is redundancy in a database, then all those values have to be locked while updating.
- Otherwise they may disagree with each other.

Again degrading throughput.

Definitions

Keyword	Definition
Lookup cost	The time spent searching/matching the appropriate records.
Data movement cost	The time spent sending query and receiving results.

Normal Form Representation

Definitions
- A **closure** iterates until there are no more changes.
- A unique **normal form** for an information can be defined.

To find the normal form, apply information-preserving rewrites until closure.

A **normalised database** has little or no redundant data: all values in records depends only on the primary key.

Specialised Database

Low redundancy $\implies$ Good update throughput
(few locks needed)

High redundancy $\implies$ Good query throughput
(fewer pointers needed to follow to reach the value)

Amount of redundancy can speedup read-oriented or write-oriented transactions in expense to slow down the other.

The write-oriented DB can **copy snapshots** to a denormalised, read-oriented DB for fast access. Puting a copy of the DB closer to the user reduces data movement cost.

- **Embedded databases** are created by extracing parts of the main DB that will never change, and store it on a device for local access.
- **Online analytical processing** (OLAP) supports write-only/ledger style updates, usedin data warehouses.
- **Online transaction processing** (OLTP) supports a mix of read and write queries.
- **ETL** extracts from OLTP, transforms the data, then loads it to OLAP.

Update History
The update history of a field can be stored by adding a time dimension for each field.

Semi-structured Data

A book written in English has some structure, e.g. chapters, figures. They will need to be indexed for easy retrieval.

Relational database	Document oriented database
Text stored in native form.	Paragraphs stored in native form.
Indexes in tables with strict schema.	Content can be **shredded** to many bits.

> **Note**
> Traditionally this is done using simple keyword analysis/advance NLP analysis. More recently **LLMs** are used to index content.

Associative Store (Key Value Store)
Values are stored as **blobs**.
- Massive and unintepretable sequence of bytes, e.g. strings.
- The data is **opaque** and the DB does not know what is being stored.

Associative store can use ACID or BASE depending on implementation.

> **Definition**
> **Sharding**: data is stored over multiple machines.

- When one machine fails, the data is still available.
- Provides load balancing.

Serialisation of Data

> **Definition**
> **Serialisation** converts a data structure/semi-structured document into a sequence of bytes, so it can be transmitted or stored in secondary storage, etc.

Encoding format	Description
JSON	Originally designed to serialising data structure.
XML	For serialising data, but also used as markup languages.
CSV	Represents **2D data**, used to transfer data between two DBs.

Few more quirks of XML:
- All data is contained in one document, which can contain text, or atomic values.
- An **XML schema** specifies the elements that need to exist, allowable attributes, occurances counts, etc.

Document Oriented Database

Definition
A **document oriented database** stores data as semi-structured objects, but there are some structure within the blobs.

Note
Document oriented databases are also called **aggregate oriented databases**.

A denormalised schema allows us to pull much more data with one query (fields that don't depend on the key can still be in the document). This reduces data movement cost.

Key Nestings

To have fast retrieval of data, we may precompute and store multiple key nestings.

`A` on top of tree.

```
[
  {
    "A": "a1",
    "R": [{ "B": "b1" }, { "B": "b2" }]
  },
  {
    "A": "a2",
    "R": [{ "B": "b3" }, { "B": "b4" }]
  },
]
```

`B` on top of tree.

```
[
  {
    "B": "b1",
    "R": [{ "A": "a1" }]
  },
  {
    "B": "b2",
    "R": [{ "A": "a1" }]
  },
  {
    "B": "b3",
    "R": [{ "A": "a2" }]
  },
  {
    "B": "b4",
    "R": [{ "A": "a2" }]
  },
]
```

In this case storing replicates of the same data can make searching faster with key `A` and `B`.

The NoSQL Movement

The NoSQL movement

- Don't like types
- Likes horizontal scaling - adding more machines instead of upgrading a single machine. This is usually cheaper and uses less power.

But types can be useful.

```
type velocity_t = branded float
type distance_t = branded float
```

Now there are two types, `[velocity_t] + [distance_t]` will give a type error, which is helpful.

Misc SQL Information

Definitions

Keyword	Definition
Declaration	A statement that holds at all times.
Recursion	A statement that refers to itself.

Complexity of Joins

- $R \bowtie S$ takes $O(R \times S)$ time (quadratic) using the brute force approach.
- If joined using a key, we can use an index. So lookups are done in $O(\log n)$ time instead of $O(n)$.

```
CREATE INDEX index_name on S(B);
DROP INDEX index_name;
```

Note
Index speeds up reads and slows down updates.

SQL as a Multiset

SQL queries returns a multiset:

name	age	salary
Alex	19	0
Bob	18	−1000
John	19	1000

- `SELECT age FROM table` clearly gives a multiset.
- To get a set (no duplicates) `SELECT DISTINCT age FROM table`.

Multisets are required by **aggregated function**.

`GROUP BY age` creates multiple tables.

name	age	salary
Bob	18	0

name	age	salary
Alex	19	−1000
John	19	1000

These tables then get passed to functions like `AVG(salary)`

age	avg(salary)
18	−1000
19	500

All aggregated function have the following properties:
- **Associative, commutative**. Otherwise the order of the items will affect the output.
- Takes a vector (multiple values) then return a scalar.

Null

`NULL` is not of any type, for all values `(X = NULL) = NULL`

This gives us **three valued logic**.

v	$\neg v$
T	F
F	T
$\bot$	$\bot$

$\vee$	**T**	**F**	$\bot$
T	T	T	T
F	T	F	$\bot$
$\bot$	T	$\bot$	$\bot$

$\wedge$	**T**	**F**	$\bot$
T	T	F	$\bot$
F	F	F	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$

Where $\bot$ denotes `NULL`.

Interpretations of Null

1. There is a value, but we don't know it yet.
2. No value is applicable.
3. Value is known, but we are not allowed to see it.

We have the null testing predicates:

```
foo IS NULL;
foo IS NOT NULL;
```

When using `GROUP BY` in a nullable field, nulls are considered equal and will be grouped together.

SQL Recursion

Definition

Function composition $(f \circ g)(x) = f(g(x))$ satisfies

- Given two binary relations $R \subseteq S \times T$ and $S \subseteq T \times U$
- Then $Q \circ R \subseteq S \times U$

If $(s, t_1) \in R \wedge (s, t_2) \in R \implies t_1 = t_2$, then R is a function.

Note

Joins are a generalisation of function (relation) composition, we can write

$$Q \circ R = R_{\bowtie 2=1} Q$$

Bacon Numbers

$G = (V, A)$ is a **directed graph** if V is the set of vertices and $A \subseteq V \times V$ is the set of arcs. If $(u, v) \in A$ then there is an arc from u to v.

Let

$$A^1 = A$$
$$A^{n+1} = A \circ A^n$$

Then $(v_0, v_k) \in A^k$ if there is a sequence of vertices $v_0, v_1, \ldots, v_k$ of arc length k from v_1 to v_k.

Define the distance from s_0 to s' be the smallest n such that $(s_0, s') \in R^n$.

Shortest Path Query

Without using recursion:

```
CREATE VIEW path_1 AS
  SELECT DISTINCT id2 AS id, 1 AS n
  FROM neighbours
  WHERE neighbours.id1 = 'source_id'
```

```
CREATE VIEW path_2 AS
  SELECT DISTINCT id2 as id, 2 AS n
  FROM path_1
  JOIN neighbours ON neighbours.id1 = path_1.id
  WHERE neighbours.id2 <> 'source_id' AND
        NOT (neighbours.id2 IN (SELECT id FROM path_1))

CREATE VIEW path_3 AS
  SELECT DISTINCT id2 as id, 2 AS n
  FROM path_2
  JOIN neighbours ON neighbours.id1 = path_1.id
  WHERE neighbours.id2 <> 'source_id' AND
        NOT (neighbours.id2 IN (SELECT id FROM path_1))
        NOT (neighbours.id2 IN (SELECT id FROM path_2))
```

This pattern repeats until $\text{path}_n = \text{path}_{n+1}$ (no more new nodes can be added).

```
CREATE VIEW distances AS
  SELECT *  FROM path_1
  UNION SELECT *  FROM path_2
  UNION SELECT *  FROM path_3;
```

Definition

Let $R \subseteq S \times S$, the **transitive closure** of R, denoted R^+ is the smallest binary relation on S such that $R \subseteq R^+$ and R^+ is transitive.

$$(x, y) \in R^+ \land (y, z) \in R^+ \implies (x, z) \in R^+$$

If R^+ is a transitive closure of R, then

$$R^+ = \bigcup_n R^n$$

If the relation is finite, then there exists a k such that

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

We can write the **recursive** query:

```
WITH distances AS
  (SELECT 0 AS n, id2 AS id
   FROM neighbours
   WHERE id1 = 'source_id' AND id1 = id2
   UNION SELECT n + 1 as n, neighbours.id2 as id
         FROM distances
         JOIN neighbours ON neighbours.id1 = id
         WHERE NOT (neighbours.id2 IN (SELECT id FROM distances)) AND n < 20)
SELECT n, COUNT(*)
FROM (SELECT min(n) AS n, id FROM distances GROUP BY id)
GROUP BY n
```

Graph Databases

Definition

A **graph** is a collection of nodes and edges.

- Entities are represented by **nodes**.
- Relations are represented by **edges**.

We could store a graph in a relational database, but they are inefficient as:
- All edges must be scanned to find neighbour of a node.
- Recursive SQLs are painful, graph databases have them built-in.
- Ends of an edge are interchangeable:
  - Both fields need to be examined for undirected search.
  - Two inverted indexes needed to be stored.

Neo4J Queries

Create nodes and edges.

```
CREATE (id123456:Entity { name: 'Name' }) // new node
CREATE (id123456) -[:RELATION_NAME { property: 'Property' }]-> (id234567) // new edge
```

Patterns

```
(a) --> (b) // all pairs of nodes from one to another
(a) -- (b) // all pairs with an edge between them
() --> (a) <-- () // any node with two or more incoming edge
(a:Person) --> (b:Movie) // connection from a Person to Movie
(a) -- (b) -- (c) --> (d) // four connected nodes (a can equal c)
() -[:ACTED_IN]-> (b) // nodes with an incoming ACTED_IN edge
(a) -[:ACTED_IN]-> (b) // all pairs related by ACTED_IN
(a) -[:ACTED_IN*]-> (b) // transitive matching
(a) -[*3..5]-> (b) // all pairs a and b separated by 3 to 5 edges
```

Querying with Patterns

```
MATCH [pattern]
RETURN a.name, b.name
```

The Bacon number query:

```
MATCH paths=allshortestpaths(
  (m: Person { name: 'Kevin Bacon' }) -[:ACTED_IN*]- (n:PERSON))
WHERE m.person_id <> n.person_id
RETURN length(paths)/2 AS bacon_number,
       COUNT(distinct n.person_id) AS total
ORDER BY bacon_number;
```

END Databases