Queens' College Cambridge

Databases

Alistair O'Brien

Department of Computer Science

April 19, 2020

Contents

1 Databases

1.1 Databases

Definition 1.1.1. (**Database**) A **database** is an organised collection of related data, stored and formatted such that data can be accessed quickly.

Definition 1.1.2. (**Database Management Systems**) A DBMS is a collection of related programs for defining, creating, maintenance and manipulation of a database.

- A query engine implements the interface between DBMS and database. Query engine uses low level details for automatic query optimisation.
- Main functions of DBMS:
  - Persistent Storage.
  - Allows concurrency.
  - Controls database access.
  - Provides query language and query optimisation.
  - Implements CRUD and ACID.

1.1.1 CRUD and ACID

- **Create**: Insert new data items into the database.
- **Read**: Query the database.
- **Updated**: Modify objects in the database.
- **Delete**: Remove data from the database.

These operations should be carried out using ACID transactions:

- **Atomicity**: Either all of the transaction are carried out, or none are.
- **Consistency**: Every transaction applied to a consistent database leaves it in a consistent state.
- **Isolation**: Transactions are isolated, or protected, from the effects of other concurrently executed transactions.
- **Durability**: If a transaction completes successfully, then it effects persist.

1.1.2 Data Models

1. **Relational Model**: Data is stored in tables. SQL is the main query language. An example is MySQL, HyperSQL, etc.
2. **Graph-oriented Model**: Data is stored as a graph (vertices and edges). Query languages tend to be "path-oriented" capabilities. An example is Neo4j, using the query language Cypher.
3. **Aggregate-oriented Model**: Often referred to as document-oriented databases. They're optimised for read-oriented databases. An example is DOCtorWho. Stores JSON objected. Query language is Python. (Implemented using a key-value store).

- Different models $\implies$ different **trade-offs**.

1.1.3 Redundant Data

Definition 1.1.3. (**Redundant data**) Data in a database is redundant if it can be deleted and then reconstructed from the data remaining in the database.

- Read / aggregate orientated databases:
  - Redundancy is desirable.
  - Low redundancy $\implies$ computing complex queries (including many joins) which is very slow
  - Redundancy $\implies$ pre-computing queries, which increases query response time.

- Write orientated databases:
  - Redundancy is undesirable.
  - Introduces inconsistencies.
  - When writing, redundancy $\implies$ more write operations are required. Databases may need to be locked to protect from concurrent updates.

Anomalies

- Insertion anomalies
- Update anomalies
- Deletions anomalies

2 ER Models and Diagrams

2.1 The ER Model

Definition 2.1.1. (**Attribute Type**) An attribute type $\overline{a}$ is a set of all possible attributes with attribute name n and value v such that v is a member of the set of possible attribute values $\operatorname{dom} a$. So

$$\overline{a} = \{n\} \times \operatorname{dom} a.$$

Definition 2.1.2. (**Attribute**) An attribute is any characteristic of an entity or relationship. An attribute a of attribute type $\overline{a}$ must satisfy $a \in \overline{a}$

Definition 2.1.3. (**Entity Type**) An entity type $\overline{E}$ can be defined as a collection of attributes types $\overline{a}_1, \overline{a}_2, \ldots, \overline{a}_n$, such that $\overline{E} = (\overline{a}_1, \overline{a}_2, \ldots, \overline{a}_n)$.

Definition 2.1.4. (**Entity Set**) An **entity set** E is a collection of entity instances $e_1, e_2, \ldots, e_n$ of some entity type $\overline{E}$. We will define a **universal entity set** E_U of some entity type $\overline{E}$ as

$$E_U = \overline{a}_1 \times \overline{a}_2 \times \cdots \times \overline{a}_m.$$

Then it follows that

$$E \subseteq E_U.$$

2.2 Relationships

Definition 2.2.1. (**Relationship Type**) A relationship type $\overline{R}$ among n entity types $\overline{E}_1, \overline{E}_2, \ldots, \overline{E}_n$ which defines a set of associations among entities from these entity types with m additional relationship attribute types $\overline{a}_1, \overline{a}_2, \ldots, \overline{a}_m$ is

$$\overline{R} = (\operatorname{key} \overline{E}_1, \ldots, \operatorname{key} \overline{E}_n, \overline{a}_1, \ldots, \overline{a}_m).$$

Definition 2.2.2. (**Relationship Set**) A **relationship set** R is a collection of relationship instances r_i of some relationship type $\overline{R}$. We define the **universal relationship set** R_U of some relationship type $\overline{R}$ as

$$R_U = \text{key}\,\overline{E}_1 \times \cdots \times \text{key}\,\overline{E}_n \times \overline{a}_1 \times \cdots \times \overline{a}_m.$$

and so

$$R \subseteq R_U.$$

Definition 2.2.3. (**Relationship**) We define a **relationship** among n entities with universal sets $E_{U,1}, \ldots, E_{U,n}$ as a mapping from the involved entities to the relationship set, hence

$$R : E_{U,1} \times \cdots \times E_{U,n} \to R_U.$$

- The **degree** of a relationship type is defined as the number of participating entity types. e.g. binary, ternary, etc.
- Some relationships can be *recursive* such as friends. $f : P \times P \to F$.

Relationship Constraints

Let us consider a relationship R between entity sets S and T. We have the types:

- **One-to-one**: Every member of S is related to at most one entity of T and every member of T is related to at most one entity of S.
- **One-to-many**: Every member of T is related to at most one entity of S.
- **Many-to-many**: No constraint.

A Remark on Relationship Attributes

The attributes of one-to-one or one-to-many relationships can be migrated to one of the participating entity types.

1. For one-to-one relationship type, attributes can be migrated to either of the entity types.

2. For one-to-many relationship type, the relationship attribute can be migrated only to the entity type on the many side of the relationship.

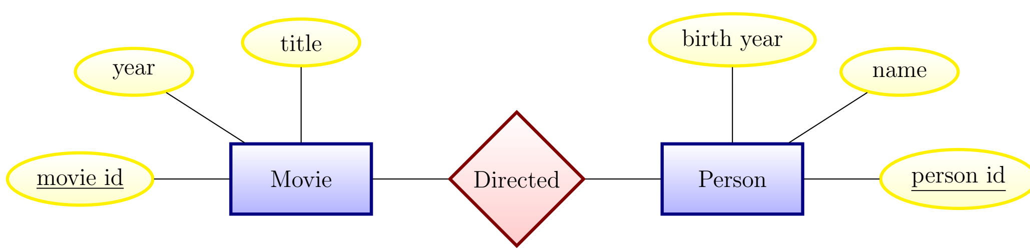
3. Attributes on many-to-many relationship types must be specified as relationship attributes.

2.3 ER Diagrams

- Three main blocks in ER diagrams:
  - Entities types (Henceforth entities). Represented as a rectangle with the entity name (a noun).
  - Relationships types (relationships). Represented as a diamond with the relationship name (a verb).
  - Attributes types (attributes). Represented as an oval with an attribute name.

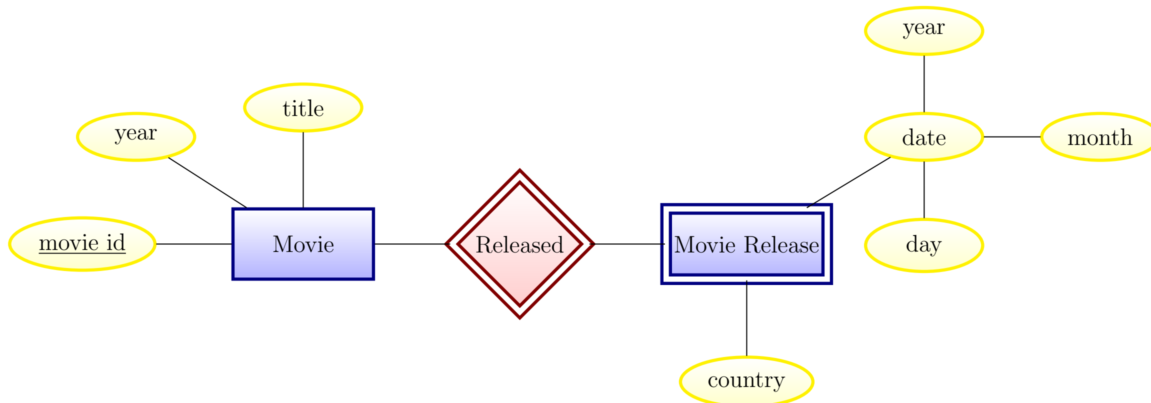
    The attribute A is an attribute of entity E if there exists an edge between them. Key attributes are underlined.

    Attributes can have sub-attributes. Known as a **multi-attribute**.



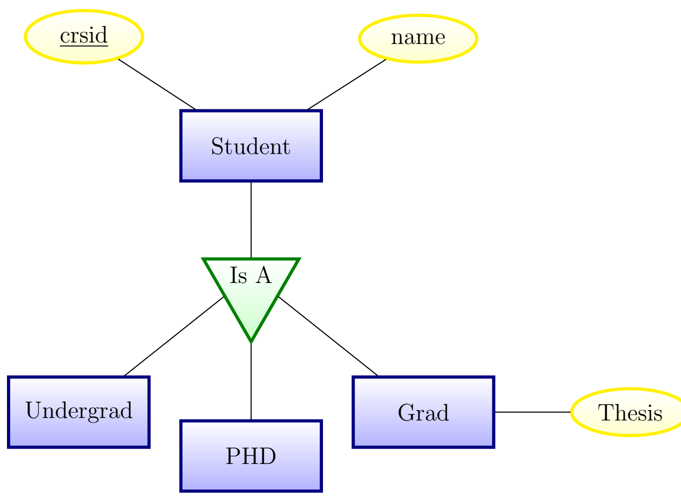
Definition 2.3.1. (**Weak entities**) Weak entities are defined as entity types $\overline{E_1}$ such that the existence of an entity instance $e_{1,i}$ of type $\overline{E_1}$ depends on the existence of another entity instance $e_{2,j}$ of type $\overline{E_2}$.

- e.g. a release date only exists in relation to an existing movie.



2.3.1 Entity Hierarchy

Sometimes an entity can have "sub entities". For example



- Entities can have sub-entities.
- Sub-entities inherit the attributes and relationships of the parent entities, however the sub-entities may have relationships and attributes specific to that sub-entity.

2.3.2 Benefits of ER diagrams

- Forces us to think clearly about the model we wish to implement without going into database-specific details (e.g. document, relational or graph data models).

- Simple and easy to learn.
- Very valuable in developing a model in collaboration with clients who know nothing about database implementation details.

3 The Relational Algebra

Definition 3.0.1. (**Relation**) A relation R over a collection of sets (domains) $S_1, S_2, \ldots, S_n$ is

$$R \subseteq S_1 \times S_2 \times \cdots \times S_n.$$

Thus a relation is a set of n-tuples $(s_2, s_2, \ldots, s_n)$ where $s_i \in S_i$.

Definition 3.0.2. (**Relation Schema**) Let $A_1, A_2, \ldots, A_n$ be attribute names with associated domains $S_1, S_2, \ldots, S_n$, then

$$R(A_1 : S_1, A_2 : S_2, \ldots, A_n : S_n).$$

is said to be the **relation schema**.

Definition 3.0.3. (**Relation Instance**) A relation instance $r(R)$ of a relation schema R can be thought of as a table with n columns and m rows.

1	2	$\cdots$	n
$s_{1,1}$	$s_{1,2}$	$\cdots$	$s_{1,n}$
$s_{2,1}$	$s_{2,2}$	$\cdots$	$s_{2,n}$
$\vdots$	$\vdots$		$\vdots$
$s_{m,1}$	$s_{m,2}$	$\cdots$	$s_{m,n}$

An element $t \in r(R)$ is referred to as a tuple (or row). Instead of relation instance, we often just say relation. An instance of a database schema thus is just a collection of relations. Note that $r(R)$ is not a multi-set.

3.1 Key Constraints

Definition 3.1.1. (**Key**) A set of attributes is a *key* for a relation schema R if no two distinct tuples have the same values for all key attributes.

- $R(\mathbf{X})$ is a relational schema with $\mathbf{Z} \subseteq \mathbf{X}$. If for all tuples u and v in any instance of $r(R)$

$$u.[\mathbf{Z}] = v.[\mathbf{Z}] \implies u.[\mathbf{X}] = v.[\mathbf{X}],$$

  then $\mathbf{Z}$ is a superkey for R.

- If no proper subset of $\mathbf{Z}$ is a superkey, then $\mathbf{Z}$ is the key for R.

Definition 3.1.2. (**Foreign Key**) Suppose that $R_1(\underline{\mathbf{Z}}, \mathbf{Y})$ and $R_2(\mathbf{W})$ be a relational schema with $\mathbf{Z} \subseteq \mathbf{W}$. We say that $\mathbf{Z}$ represents a **foreign key** in R_2 for R_1 if for any instance we have

$$\pi_{\mathbf{Z}}(R_2) \subseteq \pi_{\mathbf{Z}}(R_1).$$

Definition 3.1.3. (**Referential Integrity**) A database is said to have **referential integrity** when all foreign key constraints are satisfied.

3.2 The Relational Algebra

- The relational algebra Q is a procedural query language, where queries are applied to relation instances $R_1, R_2, \ldots, R_n$ and results in a single relation instance R.

$$Q : R_1 \times R_2 \times \cdots \times R_n \to R.$$

- Syntax:

$$\begin{aligned} Q ::= {} & R && \text{Relation instance} \\ \mid {} & \sigma_p(Q) && \text{Selection} \\ \mid {} & \pi_X(Q) && \text{Project} \\ \mid {} & Q \times Q && \text{Product} \\ \mid {} & Q - Q && \text{Difference} \\ \mid {} & Q \cup Q && \text{Union} \\ \mid {} & Q \cap Q && \text{Intersection} \\ \mid {} & \rho_M(Q), && \text{Renaming} \end{aligned}$$

  where

1. p is a Boolean predicate over attribute values
2. $X = \{A_1, A_2, \ldots, A_k\}$ is a set of attributes.
3. $M = \{A_1 \mapsto B_1, A_2 \mapsto B_2, \ldots, A_k \mapsto B_k\}$ is a renaming map.

Definition 3.2.1. (**Well formed**) A query Q is said to be well-formed if and only if all the column / attribute names of the result are distinct.

3.2.1 Selection

Definition 3.2.2. (**Selection**) The selection operation $\sigma_p(R)$ on a relation instance is defined as

$$\sigma_p(R) = \{t \in R : p(t)\},$$

where R is a relation instance and p is a Boolean predicate over the attributes.

The corresponding SQL query for the selection operation is

```
SELECT DISTINCT * FROM R WHERE p;
```

where `p` is some Boolean predicate on the attributes of `R`.

3.2.2 Projection

Definition 3.2.3. (**Projection**) The projection operation $\pi_{A_1,A_2,\ldots,A_k}(R)$ on a relation instance R is defined as the relation that has k columns obtained by removing all columns from R that aren't in the set $\{A_1, A_2, \ldots, A_n\}$ and then removing any duplicate rows.

The corresponding SQL query for the project operation is

```
SELECT DISTINCT A_1, ..., A_n FROM R;
```

3.2.3 Renaming

Definition 3.2.4. (**Renaming**) The renaming operation $\rho_{\{A_1 \mapsto B_1, \ldots, A_k \mapsto B_k\}}(R)$ on the relation instance R maps the names of the column $A_1, \ldots, A_k$ to $B_1, \ldots, B_k$.

The corresponding SQL query for the renaming operation is

```
SELECT A_1 AS B_1, ..., A_k AS B_k, A_{k + 1}, ..., A_n FROM R;
```

3.2.4 Union

Definition 3.2.5. (**Union**) The union operation $R \cup S$ on the relation instances R and S is defined as

$$R \cup S = \{t : t \in R \vee t \in S\}.$$

For $R \cup S$ to be applicable:

- R and S must have the same number of attributes
- Attribute domains must be compatible. e.g. 3rd column of R must have the same data type as the 3rd column of S.

The corresponding SQL query for the union operation is

```
(SELECT * FROM R) UNION (SELECT * FROM S);
```

3.2.5 Difference

Definition 3.2.6. (**Difference**) The difference operation $R \cap S$ on the relation instances R and S is defined as

$$R \cup S = \{t : t \in R \wedge t \in S\}.$$

For $R \cap S$ to be applicable:

- R and S must have the same number of attributes
- Attribute domains must be compatible

The corresponding SQL query for the difference operation is

```
(SELECT * FROM R) EXCEPT (SELECT * FROM S);
```

3.2.6 Product

Definition 3.2.7. (**Product**) The product operation $R \times S$ on the relation instance R and S is defined as

$$R \times S = \{t + q : t \in R \wedge q \in S\}.$$

where $+ : K^n \times K^m \to K^{m+n}$ is a binary operations that appends one tuple to another.

For $R \times S$ to be applicable:

- $R \cap S = \emptyset$. Otherwise the renaming operation must be applied before product.

The corresponding SQL for the product operation is

```
SELECT * FROM R CROSS JOIN S;
```

3.2.7 Join Operations

Some Notation

- From henceforth, the relation schema $R(A_1 : S_1, A_2 : S_2, \dots, A_n : S_n)$ will be denoted as $R(\mathbf{A})$ where $\mathbf{A} = \{A_1, A_2, \dots, A_n\}$.

- When we write $R(\mathbf{A}, \mathbf{B})$ we mean $R(\mathbf{A} \cup \mathbf{B})$ and we assume that $\mathbf{A} \cap \mathbf{B} = \emptyset$.

- When we write $u.[\mathbf{A}] = v.[\mathbf{A}]$, we mean $u.A_1 = v.A_1 \wedge u.A_2 = v.A_2 \wedge \dots \wedge u.A_n = v.A_n$.

Definition 3.2.8. (**Natural Join**) The natural join over the relation instance $R_1(\mathbf{A}, \mathbf{B})$ and $R_2(\mathbf{B}, \mathbf{C})$ is given by

$$R_1 \bowtie R_2 = \{u.[\mathbf{A}] \cup u.[\mathbf{B}] \cup v.[\mathbf{C}] : \exists u \in R_1, v \in R_2, u.[\mathbf{B}] = v.[\mathbf{B}]\}.$$

In relational algebra, we may write this as

$$R_1 \bowtie R_2 = \pi_{\mathbf{A},\mathbf{B},\mathbf{C}} \left(\sigma_{\mathbf{B}=\mathbf{B}'} \left(R_1 \times \rho_{\mathbf{B} \mapsto \mathbf{B}'}(R_2)\right)\right).$$

- Left / Right outer-joins performs a natural join but also includes all of the rows in the left / right table that do not match with any rows from the other table, using `NULL` values to fill the empty columns.

  The left outer-join over the relations $R(\mathbf{A}, \mathbf{B})$ and $S(\mathbf{B}, \mathbf{C})$ is

$$R \bowtie_{left} S = (R \bowtie S) \cup (R - \pi_{\{\mathbf{A},\mathbf{B}\}}(R \bowtie S) \times \{(\underbrace{\perp, \dots, \perp}_{|\mathbf{C}| \text{ times}})\}).$$

- Full outer-join is the union of the left and right outer-joins.

3.3 Relations

Definition 3.3.1. (**Composition**) The composition over binary relations $R \subseteq A \times B$ and $S \subseteq B \times C$ is given by

$$R \circ S = \{(a, c) : \exists b \in B : (a, b) \in R \wedge (b, c) \in S\}.$$

- A (partial) function $f \in A \to B$ can be thought of as a binary relation where $(a, b) \in f$ if and only if $b = f(a)$.
- Suppose R is a relation such that $A \cup B$ is the key of R, then R represents a (partial) function.
- Given two partial functions $f \in A \to B$ and $g \in B \to C$, their composition $g \circ f \in A \to C$ is $(g \circ f)(a) = g(f(a))$
- Note that
  $$R \circ S = \rho_{\{1,3\}}(R \bowtie S).$$
  So joins are a generalises of composition.

4 Graph Databases

4.1 The Flaws Of SQL

4.1.1 Joins

- $R \bowtie S$ is naively implemented as

```
for each (a, b) in R do
    for each (b', c) in S do
        if b = b' then
            append(result_set, (a,b,c))
        end
    end
end
```

- Has time complexity of $O(|R| \times |S|) \implies$ inefficient.
- **Solution** : Indexes.
  - An **index** is a data structure (such as a hash-map or binary tree) that reduces the time required to locate records.

```
for each (a, b) in R do
    for each s in S_INDEX_B(b) do
        append(result_set, (a,b,s.c))
    end
end
```

  - SQL commands:

```
CREATE INDEX IF NOT EXISTS index_name
    ON table_name(column_name);

DROP INDEX index_name;
```

  - Speed up `JOIN`s and queries, but slow down updates (bcs redundancy).

4.1.2 NULLS and Three Valued Logic

- `NULL` is a place-holder, not a value.
- It's not a member of any domain / type.
- We require three-valued logic:

$$\begin{array}{c|ccc} \wedge & T & F & \bot \\ \hline T & T & F & \bot \\ F & F & F & F \\ \bot & \bot & F & \bot \end{array}$$

$$\begin{array}{c|ccc} \vee & T & F & \bot \\ \hline T & T & T & T \\ F & T & F & \bot \\ \bot & T & \bot & \bot \end{array}$$

$$\begin{array}{c|c} a & \neg a \\ \hline T & T \\ F & F \\ \bot & \bot \end{array}$$

- `NULL` is ambiguous in SQL and has many possible interpretations.
- Inconsistent logic: e.g. $E_1 = E_2$ will evaluate to `NULL` if E_1 or E_2 are `NULL`.

4.2 Graphs and Graph Databases

4.2.1 Graphs

Definition 4.2.1. (**A Graph**) A graph G consists of a finite, non-empty set of vertices V and a binary relation of edges E over V × V.

- Every edge $e = (u, v) \in \mathrm{E}$ joins two u, v vertices, called **endpoints**. u, v are **adjacent**

Definition 4.2.2. (**An Undirected Graph**) An **undirected graph** G is a graph in which the edge set $\mathrm{E}(G)$ consists of unordered pairs.

Definition 4.2.3. (**A Directed Graph**) A **directed graph** G is a graph in which the edge set $\mathrm{E}(G)$ consists of ordered pairs. The term "directed graph" is often abbreviated as **digraph**.

4.2.2 Iterated Composition, Paths and Transitive Closure

Definition 4.2.4. (**Iterated Composition**) Suppose R is a binary relation over S, such that $R \subseteq S \times S$. We define **iterated composition** as

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

Let $G = (\mathrm{V}, \mathrm{E})$ be a directed graph. Suppose we have a (v_1, v_k)-path in G of length k, then it follows that $(v_1, v_k) \in \mathrm{E}^k$

Definition 4.2.5. (**R-distance**) Suppose we have the binary relation R over S, $R \subset S \times S$ where $s_0 \in \pi_1(R)$, that it to say that $(s_0, s_1) \in R$. Then

- The distance form s_0 to s_0 is zero.
- If $(s_0, s_1) \in R$, then the distance from s_0 to s_1 is 1.
- For any other $s' \in \pi_2(R)$, the distance from s_0 to s' is the least k such that $(s_0, s') \in R^k$.

Definition 4.2.6. (**Transitive Closure**) Suppose R is a binary relation over S, $R \subset S \times S$. The **transitive closure** of R, denoted R^+, is the smallest binary relation on S such that $R \subset R^+$ and R^+ is **transitive**

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

So

$$R^+ = \bigcup_{n \in \mathbb{Z}^+} R^n.$$

For finite relations R, there there exists some k such that

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

However k will depend on R, so it follows that the transitive closure cannot be computed using Relational Algebra (or SQL without recursion).

Bacon Number

4.2.3 Graph Databases

- A graph database is a database that uses a directed graph for queries with vertices, edges and properties that store data.
- Vertices and edges store properties represented by key-value pairs.
- Vertices and edges are grouped by *labels*.

Cypher Syntax

- **Read Query Structure**:

```
[match where]+
[optional match where]+
return [order by] [limit]
```

- **Match**:
  - Node patterns can contain labels and properties `match (n:Person {name: "Alice"})`.
  - Assign a path to `p` `MATCH p = <pat>`

- Optional pattern: nulls used for missing parts optional match (n) -[r]-> (m)

- **Where**:
  - where <predicate>
  - Predicates:
    - Comparison operators: =, <>, >, <, >=, <=
    - Logical operators: AND, NOT, OR
    - Label check: n:<labels>
    - Check if something is null: IS NULL, IS NOT NULL
    - Property in list: n.property IN [x1, x2, .., xn]
    - Pattern matching: e.g. (has edge) (n)-->(m).

- **Return**:
  - Return the value of all variables: return *
  - Alias: return n as name
  - Return unique rows: return distinct n
  - Sort the result order by n.property (DESC|ASC), ...
  - Limit the number of results limit 10.

- **Patterns**:
  - Labels (n:label1:label2 ... labeln)
  - Node with properties (n:label {k: v})
  - Relationships ()-[r:label {k: v}]-()
  - Directed relationship (n)-[r:label]->(m) from n to m.
  - Multiple relationships (n)-[r1:label1|r2:label2]-(m)
  - Variable length path of between min and max from n to m: (n)-[r:label*min..max]-(m)
  - Find all shortest paths allShortestPaths(<pat>)

- **Aggregate Functions**:

- Aggregate functions implicitly group by on the other non-aggregate return expressions (grouping keys). e.g. `return n, count(*)` groups by `n`.
- `count(variable)` counts the number of non-null values.
- `collect(n.property)` list from values `n` (ignores null properties).
- `sum(n.property)`, (similar to `avg, min, max`). Requires numerical value.

5 Aggregate-Oriented Databases

5.1 Semi-Structured Data

Definition 5.1.1. (**Semi-structured data**) **Semi-structured** data is a form of **structured data** that does not obey the formal structure of data models associated with relational databases, but contains tags to separate elements and enforce hierarchies of records and fields within the data.

5.1.1 Types

XML

XML (Extensible Markup Language) is a markup language used for encoding documents in a "human" and machine readable format.

An example of XML is

```
<menu id="file" value="File">
    <popup>
        <menuitem value="New" onclick="CreateNewDoc()" />
        <menuitem value="Open" onclick="OpenDoc()" />
        <menuitem value="Close" onclick="CloseDoc()" />
    </popup>
</menu>
```

JSON

JSON (JavaScript Object Notation) is a standard form that uses human readable text to transmit data objects consisting of attribute-value pairs.

An example of JSON is

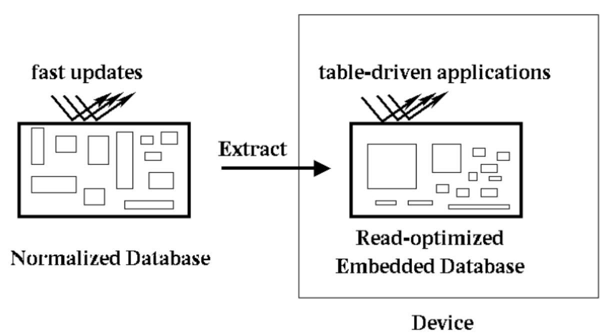
```
{"menu": {
    "id": "file",
```

```
        "value": "File",
        "popup": {
            "menuitem": [
                {"value": "New", "onclick": "CreateNewDoc()"},
                {"value": "Open", "onclick": "OpenDoc()"},
                {"value": "Close", "onclick": "CloseDoc()"}
            ]
        }
    }}
```

5.2 Aggregate-Oriented Databases

Definition 5.2.1. (**Aggregate-Oritented Databases**) A **aggregate-oriented database** (or **document-oriented databases**) stores data in the form of **semi-structured objects**.

- Semi-structured data can introduce data redundancy. So useful in read-oriented databases.
- Situations that use read-oriented databases:
  - Data stored is rarely updated, but very often read
  - Reads can be out-of-sync with the write-oriented database. Then consider periodically extracting read-oriented snapshots and storing them in a database system optimized for reading.



FIDO = Fetch Intensive Data Organization

5.3 Data Cubes