Fundamentals of Databases notes

Todd Davies

January 4, 2015

Introduction

Databases are core, if largely invisible, components of modern computing architectures in both commercial and scientific contexts. The management of data has evolved from application-specific management of myriad files to organisation-wide approaches that see data as one of the most important assets of modern organisations and, as such, a key factor in their ability to compete and thrive. At this organisation-wide scale, database management systems (DBMSs) are the crucial piece of software infrastructure needed to achieve the desired results with consistent quality and robust efficiency. A modern DBMS is a thing of wonder and embodies in its internal construction and in its wide usability many advances in algorithms and data structures, programming language theory, conceptual modelling, concurrency theory, and distributed computing. This makes the study of databases a data-centric traversal of many of the most exciting topics in modern computing.

Aims

The aim of this course unit is to introduce the students to the fundamental concepts and techniques that underlie modern database management systems (DBMSs).

The course unit studies the motivation for managing data as an asset and introduces the basic architectural principles underlying modern DBMSs. Different architectures are considered and the application environments they give rise to.

The course unit then devotes time to describing and motivating the relational model of data, the relational database languages, and SQL, including views, triggers, embedded SQL and procedural approaches (e.g., PL/SQL).

The students learn how to derive a conceptual data model (using the Extended Entity Relationship paradigm), how to map such a model to target implementation model (for which the relational model is used), how to assess the quality of the latter using normalisation, and how to write SQL queries against the improved implementation model to validate the resulting design against the data requirements originally posed. For practical work, the Oracle DBMS is used.

The course unit also introduces the fundamentals of transaction management including concurrency (e.g., locking, 2-phase locking, serialisability) and recovery (rollback and commit, 2-phase commit) and of file organisation (e.g., clustering) and the use of indexes for performance.

Finally, the course unit addresses the topic of database security by a study of threats and countermeasures available.

In the case of the former, these include potential theft and fraud as well as loss of confidentiality, privacy, integrity and availability. In the case of the latter these primarily include mechanisms for authorization and access control, including the use of views for that purpose. The course unit also addresses the topic of legal frameworks that give rise to obligations on the part of database professionals by introducing exemplars such as the 1995 EU Directive on Data Protection and the 1998 UK Data Protection Act.

Additional reading

Contents

1 An introduction to Database Management Systems

2 Relational Algebra and SQL
- 2.1 Selection σ . . . . .
- 2.2 Projection π . . . . .
- 2.3 Product $\times$ . . . . .
- 2.4 Renaming ρ . . . . .
- 2.5 Join $\bowtie$ . . . . .
- 2.6 Distinct δ . . . . .
- 2.7 Chaining operators . . . . .

3 SQL Syntax
- 3.1 Selecting rows . . . . .
  - 3.1.1 Where conditions . . . . .
- 3.2 Set operations . . . . .
- 3.3 Joins . . . . .
- 3.4 Renaming columns . . . . .
- 3.5 Ordering and other operations . . . . .

4 Designing databases
- 4.1 Entity Relationship Modelling . . . . .
  - 4.1.1 ER Notation . . . . .
  - 4.1.2 Specialising entities . . . . .

1 An introduction to Database Management Systems

DataBase Management Systems (DBMS's) are a type of middleware that provide a layer of abstraction for dealing with databases. It is nearly always unnecessary to write software from scratch that interfaces with a database, since a lot of database operations will share a significant amount of logic.

Henceforth, a lot of the functionality required of applications that make use of a database is placed into a DBMS, which application developers can make use and save time. The DBMS acts as a service, that is well implemented and is able to enforce good practices and advanced techniques such as concurrency, sharding, recovery management and transactions.

Some advantages of using a DBMS include:

- It decouples data inside a database from the application using it. Either can be re-written at any time so long as they still provide/use the same interface.
- Since the data is decoupled from the application, using a DBMS (in theory) lowers the development cost of the application.
- Most DBMS are scalable, concurrent, fault tolerant, authorisation control (often role based for organisations).

Even though the DBMS aims to provide a layer of abstraction for a user application, there are several layers of abstraction within the DBMS itself. These are:

Physical	Deals with the file(s) that is written to the storage medium that will hold the database. Needs to know about file formats, indexing, compression, etc.
Logical	Mainly concerned with mapping the raw data into database 'concepts' such as tables, views etc. It is here that the formal specification of the database is defined, commonly used models include *relational, XML based and document based*
View	Ensures that only authorised people can view the data.

If the database is using a relational format, then it will be defined by a schema. A schema dictates how the database is formatted; what tables there are, and what datatypes their columns take. An instance of a database is the content (data) inside of the database at a particular point in time. There is a certain isomorphism between relational databases and imperative programming languages; a schema would be akin to the declaration of variables (i.e. their names and types), while the instance would be their values at a particular point in the program's execution.

Irrespective of what logical model a database uses, most DBMS use between one and three languages to interface with a user/application. These are:

- Data Definition Language - used for specifying schema.
- Data Manipulation Language - used for mutating the data in the database.
- Data Query Language - used to access data in the

database.

Often DBMS languages will be both a DML and a DQL, and sometimes a DDL too! One such example is SQL, does all of the above!

2 Relational Algebra and SQL

Relational algebra is designed for modelling data stored in relational databases (i.e. tables) and defining queries on it. It can perform unary operations (such as growing, shrinking and selecting from tables), or binary operations (union, intersection, difference, product, join).

2.1 Selection σ

The σ operator can select rows that meet a certain criteria from the table.

Alcohol-Selection

Type	Strength	Colour
Wine	11	Red
Beer	4.2	Yellow
Wine	12.8	White
Port	18	Carmine
Ale	11	Red

If we run $\mathit{Fine\text{-}Wines} := \sigma_{Type=Wine}(\mathit{Alcohol\text{-}\ Selection})$, we'll end up with:

Fine-Wines

Type	Strength	Colour
Wine	11	Red
Wine	12.8	White

2.2 Projection π

The π operator can select rows instead of columns. If we do $\textit{Anonymous-Drinks} := \pi_{\textit{Strength,Colour}}(\textit{Alcohol-Selection})$:

Anonymous-Drinks

Strength	Colour
11	Red
4.2	Yellow
12.8	White
18	Carmine

Notice that both the 11% Ale and the 11% Red Wine have the same strength and colour values. Consequently, the projection operator combines those rows into one so the same result isn't displayed twice.

Projection can also be used to do simple arithmetic, $\textit{Test} := \pi_{\textit{Strength+Strength->DStrength,Colour}}(\textit{Anonymous-Drinks})$:

Test

DStrength	Colour
22	Red
8.4	Yellow
25.6	White
36	Carmine

2.3 Product ×

Shops

Name	Dodginess	Price
Ali's	High	Medium
Tesco	Low	Medium
New Zeland Wines	X.High	Low

We could do a cross product with the Shops and the Alcohol-Selection tables *Grog-Shops* := *Shops*×*Alcohol-Selection*:

Grog-Shops

Name	Dodginess	Price	Type	Strength	Co
Ali's	High	Medium	Wine	11	Re
Ali's	High	Medium	Beer	4.2	Ye
Ali's	High	Medium	Wine	12.8	W
Ali's	High	Medium	Port	18	Ca
Ali's	High	Medium	Ale	11	Re
Tesco	Low	Medium	Wine	11	Re
Tesco	Low	Medium	Beer	4.2	Ye
Tesco	Low	Medium	Wine	12.8	W
Tesco	Low	Medium	Port	18	Ca
Tesco	Low	Medium	Ale	11	Re
New Zeland Wines	X.High	Low	Wine	11	Re
New Zeland Wines	X.High	Low	Beer	4.2	Ye
New Zeland Wines	X.High	Low	Wine	12.8	W
New Zeland Wines	X.High	Low	Port	18	Ca
New Zeland Wines	X.High	Low	Ale	11	Re

2.4 Renaming ρ

The notation for renaming columns is pretty simple; $Drinks := \rho_{Name,Strength,Hue}(Alcohol\text{-}Selection)$

Drinks

Name	Strength	Hue
Wine	11	Red
Beer	4.2	Yellow
Wine	12.8	White
Port	18	Carmine
Ale	11	Red

2.5 Join $\bowtie$

If we had:

People

Name	Drinks
Alice	Wine
Bob	Beer

We could join it with the Grog-Shops table, using $Fave\text{-}Shops := People \bowtie_{People.Drinks=Grog-Shops.Type} (Grog\text{-}Shops)$

		Fave-Shops		
Person.Name	Grog-Shops.Name	Dodginess	Price	Type
Alice	Ali's	High	Medium	Wine
Bob	Ali's	High	Medium	Beer
Alice	Ali's	High	Medium	Wine
Alice	Tesco	Low	Medium	Wine
Bob	Tesco	Low	Medium	Beer
Alice	Tesco	Low	Medium	Wine
Alice	New Zeland Wines	X.High	Low	Wine
Bob	New Zeland Wines	X.High	Low	Beer
Alice	New Zeland Wines	X.High	Low	Wine

If two tables have a column of the same name, then they can be joined naturally without specifying which columns to join explicitly.

2.6 Distinct δ

The δ operator will ensure that no rows are duplicated.

2.7 Chaining operators

Just like in normal algebra, you can chain operators:

$\delta(\pi_{Strength,Colour}(\mathit{Alcohol\text{-}Selection}))$

Gives:

Strength	Colour
11	Red
4.2	Yellow
12.8	White
18	Carmine

3 SQL Syntax

3.1 Selecting rows

The SQL command to select rows from a table is:

```
SELECT <column-name>
FROM <table-name>;
```

If you only want distinct values from a column, use `DISTINCT`:

```
SELECT DISTINCT <column-name>
FROM <table-name>;
```

If you want all of the columns, use `*`:

```
SELECT *
FROM <table-name>;
```

You can also do derivations:

```
SELECT salary/2
FROM <table-name>;
```

In order to specify which rows you want from the table, or you want to join two tables together, use the `WHERE` clause:

```
SELECT *
```

```
WHERE <condition>
FROM <table-name>;
```

```
SELECT *
WHERE table1.columnx = table2.columny
FROM table1, table2;
```

Use `AND` to chain multiple conditions in a `WHERE` clause:

```
SELECT *
WHERE x > 100
AND table1.x = table2.y
FROM <table-name>;
```

3.1.1 Where conditions

You can use the standard $=, <, >$ signs for comparisons between columns, but SQL also lets you use `LIKE`, `BETWEEN` and `IS NULL`:

```
SELECT *
FROM <table-name>
WHERE name LIKE '
```

```
SELECT *
FROM <table-name>
WHERE salary BETWEEN 10000 AND 12000;
```

```
SELECT *
FROM <table-name>
WHERE previous-convictions IS NULL;
```

You can also compare tuples:

```
SELECT *
FROM t1, t2
WHERE (t1.parent, t1.age) = ('Janet', t2.age);
```

3.2 Set operations

The three main set operations are `UNION, EXCEPT` and `INTERSECT` their meaning is rather self evident given their names. An example usage may be:

```
(SELECT *
FROM <table-name>)
UNION ALL
(SELECT *
FROM <table-name>);
```

3.3 Joins

We've already looked at the most common join:

```
SELECT *
WHERE table1.columnx = table2.columny
FROM table1, table2;
```

But you can also do it manually:

```
SELECT *
FROM table1 JOIN table2
USING (<column-name>);
```

Or with a `NATURAL JOIN`, which is automatic, but can only be applied when columns have identical names:

```
SELECT *
FROM table1 NATURAL JOIN table2
```

3.4 Renaming columns

You can use the `FROM` clause to rename tables:

```
SELECT *
FROM table1 as a, table2 as b
where a.col > b.col;
```

3.5 Ordering and other operations

`GROUP BY` can be used to sort rows by a column value:

```
SELECT *
FROM <table-name>
GROUP BY height;
```

Other operators such as `AVG` and `COUNT` can also be used:

```
SELECT AVG(salary)
FROM <table-name>;
```

```
SELECT COUNT(DISTINCT salary)
FROM <table-name>;
```

```
SELECT COUNT(*)
FROM <table-name>;
```

```
SELECT dept-name, AVG(salary)
FROM staff
```

```
GROUP BY dept-name
HAVING AVG(salary) > 30000;
```

4 Designing databases

Entity-Relationship (ER) Modelling is a simple, high-level conceptual modelling approach that focuses on data requirements rather than business logic. A conceptual model is what we aim to produce in the design phase of database development. It describes the format of the database, they types of entity that are used, the relationships between entities and constraints on the data.

We can translate a conceptual model into a logical schema, which is often just a language such as SQL. We can use the logical schema directly to create and manipulate our database.

4.1 Entity Relationship Modelling

There are three basic constructs in ER modelling; entity types, attribute types and relationship types.

Attribute types consist of the following:

Simple or Composite
: A simple type is atomic, whereas a composite type is made of an amalgamation of multiple other types.

Single or Multi valued
: Types can either have single values (such as an integer),

or they can have a value from discrete set of allowed values (multi-valued).

Stored or Derived
: Stored attributes are ones where the value of the attribute is directly stored by the database (e.g. licence start date), while Derived attributes are computed from the values of other attributes (e.g. licence end date = start date + 1 year).

Null valued
: Can be null.

Complex-valued
: Made of arbitrarily nested composite or multivalued attributes.

> Note:
>
> Remember, ER models don't have primary or foreign keys, they only have plain keys. Relationships are modelled explicitly in ER rather than implicitly using keys.

Keys are important in ER modelling. Entities with a key are bound by uniqueness constraints; each key must be unique within the entity. There can be more than one key in an entity type, but there can also be no key, and if this is the case, then the entity is said to be **weak**.

Carnality ratio constraints specify the ratio of one entity to another in a relationship. It can be $1:1, 1:n, n:1, n:m$.

Participation constraints specify if an entity depends on

another entity. If a participation is **total**, then every entity in the total entity must participate in some relationship, if the participation is **partial**, then some entities may not participate in any relationship. For example, student:program is total:partial.

Weak entities can only be properly identified if they are related to some other entity type, an **owner**. An **identifying relationship** is one where a weak entity has a total participation constraint with it's owner.

When we're writing a requirements specification, **bold** words are entity types (e.g. **employees, departments and projects**), *italic* words are attribute types (e.g. *name, location, phone number*), and underlined words are relationship types (e.g. employees belong to a department).

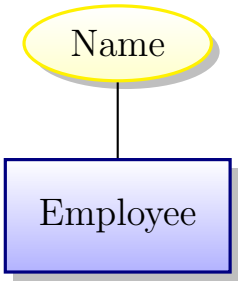
4.1.1 ER Notation

2

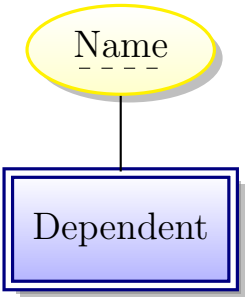
Entity type

Weak entity type

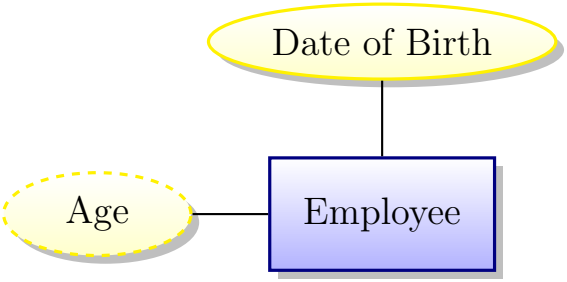
Attribute



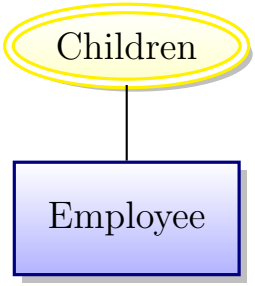
Weak attribute



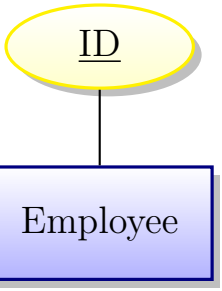
Derived attribute



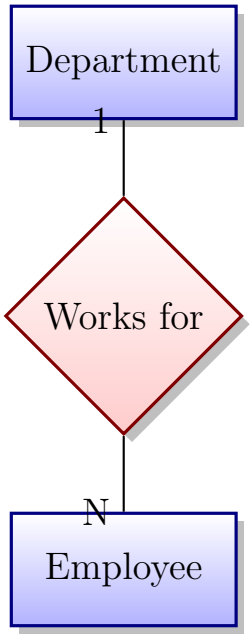
Multivalue



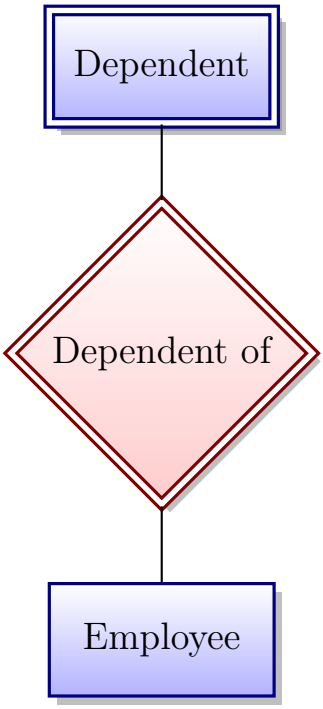
Key



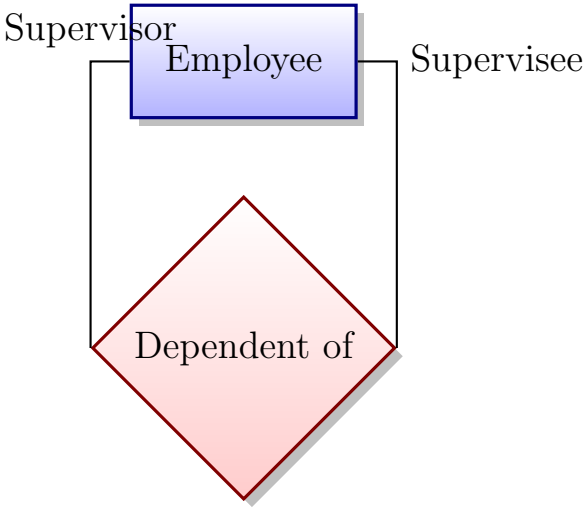
Relationship (plus carnality ratio)



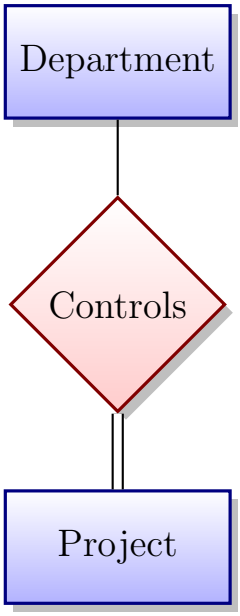
Identifying relationship



Recursive relationship



Participation constraint

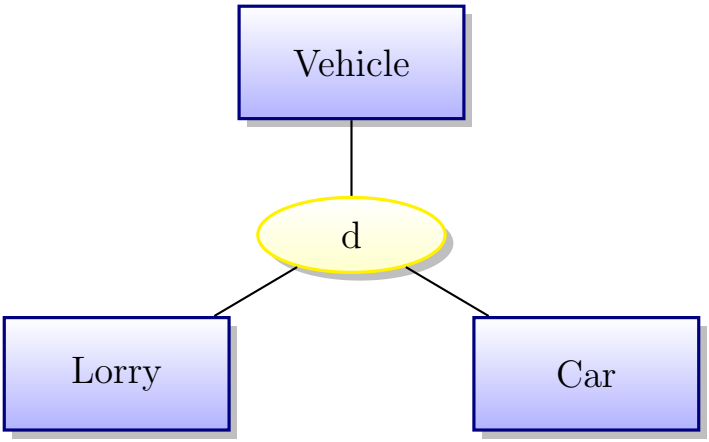


4.1.2 Specialising entities

It's often efficient to refine entities into sub-entities if there are fields that are relevant for some members and not others. For example a vehicle entity could have car and lorry sub-entities. One way of doing this is top-down conceptual refinement, which is a fancy way of saying, start with an entity, identify it's subclasses, create sub-entities for them, and then repeat for the newly created sub-entities.

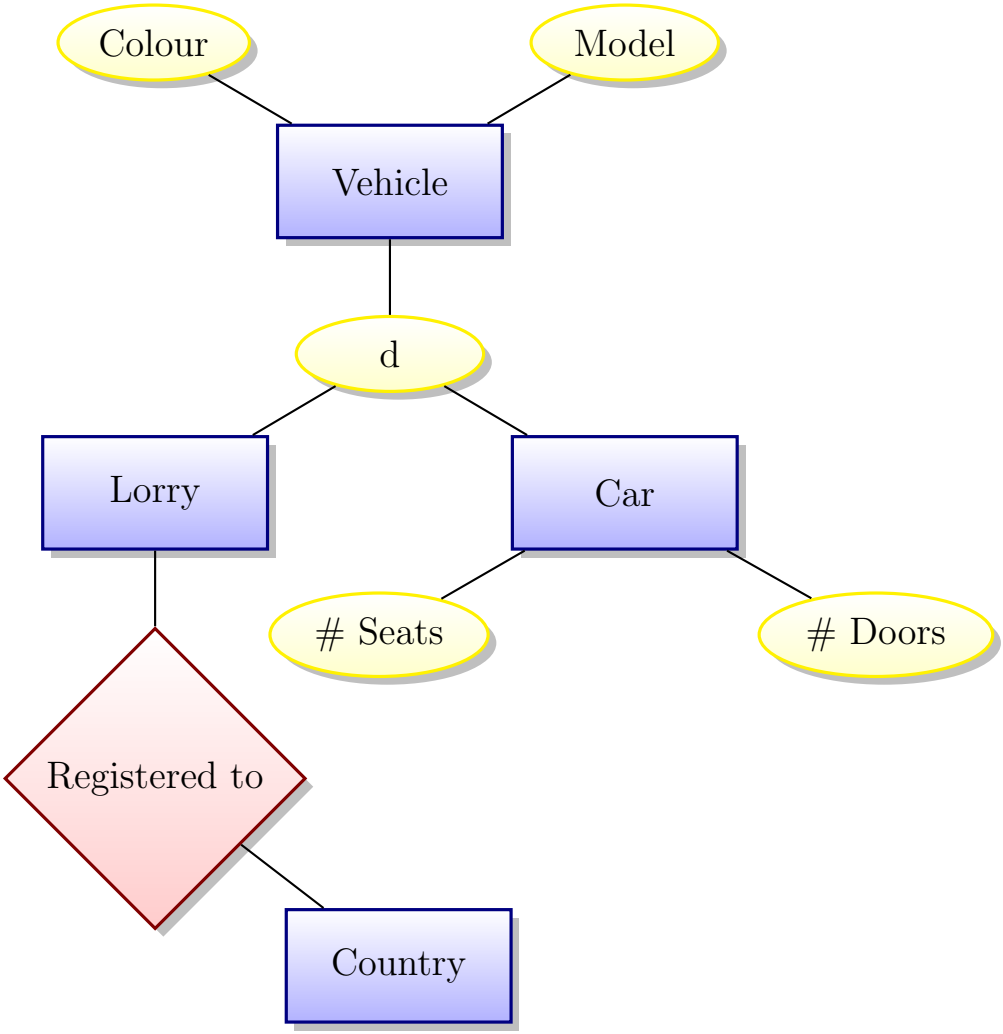
Note:

I've not got time to work out how to do $\cup$ arrowheads now, please draw them on the diagram!



An alternate way to specialise is to do a bottom-up approach. Start with lots of sets, and define a common superset of them. Here, you're generalising rather than specialising.

Each subentity can have it's own attributes and relationships of course:



Entity subtypes can be user, attribute or predicate defined based on whether the user has defined them to be a subtype, their attributes define them to be subtypes or if a predicate has (e.g. if country in EU $\implies$ EU Member).

In the examples, the attribute 'd' is used to create a subtype. This stands for disjoint, since a car can't also be a lorry. There is also 'o', which is used when an entity could belong

to multiple subtypes.