Dr Simon D’Alfonso

INFO90002 Database Systems & Information Modelling

Week 04

Physical Database Design

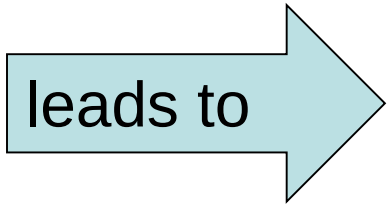
This lecture

- Physical database design
  - Estimating usage
  - Data types
  - Indexing
  - De-normalisation
- Aims of physical design
  - Translate the logical description of data into the technical specifications for storing and retrieving data on disk
  - Create a design for storing data that will provide good performance and ensure database integrity, recoverability and security
- Data Dictionaries

Physical Design Process

Inputs

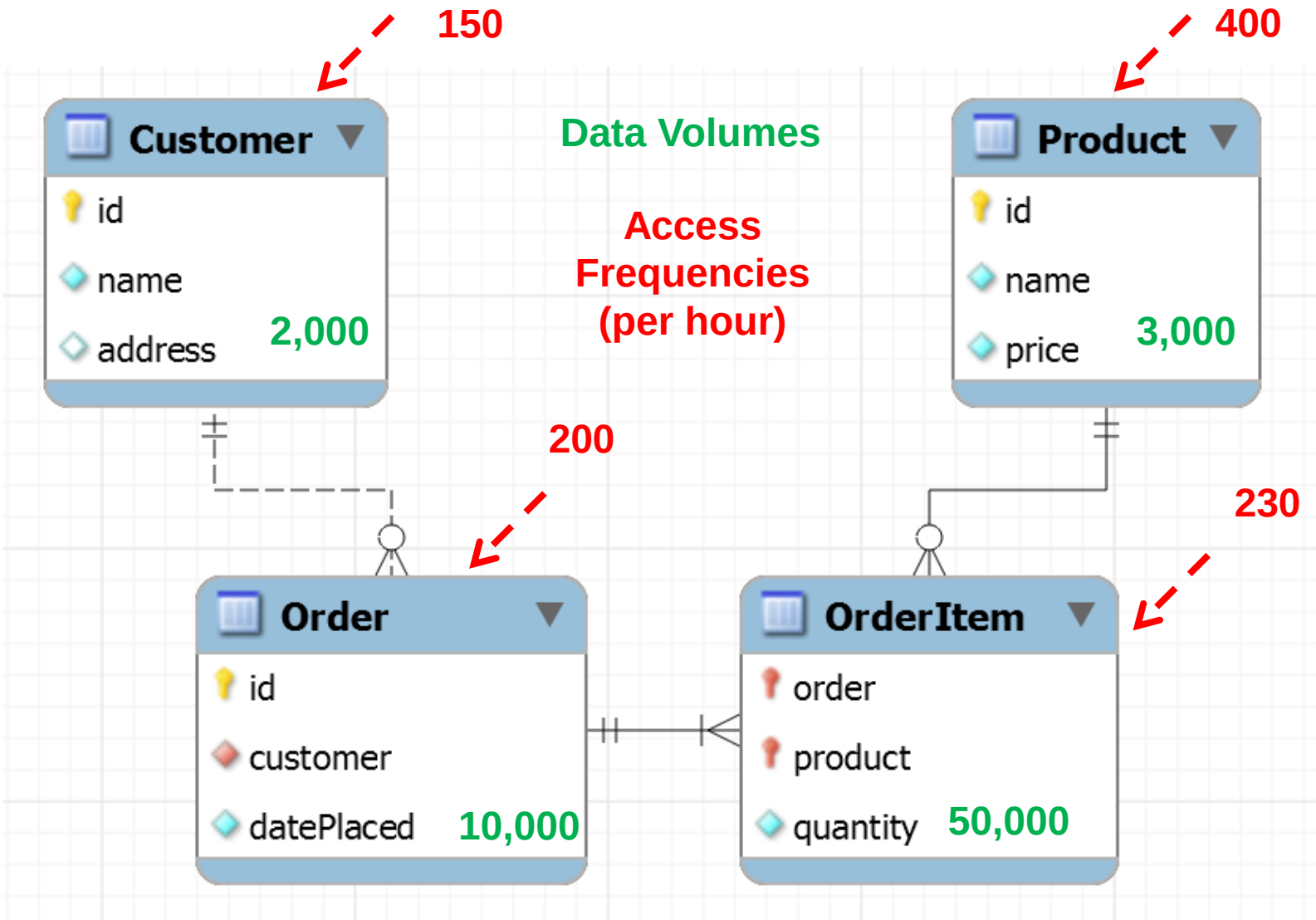
- Normalised data model
- Attribute definitions
- Volume estimates
- Response time expectations
- Data security needs
- Backup/recovery needs
- Integrity expectations
- DBMS used



Decisions

- Attribute data types
- Physical record descriptions (doesn't always match logical design)
- File organisations
- Indexes
- Query optimisation

Estimating Usage



Choosing data types

- Column: smallest unit of data in database
- Data types help DBMS to store and use information efficiently
- You should choose data types that:
  - enforce data integrity (quality)
  - can represent all possible values
  - support all required data manipulations
  - minimize storage space
  - maximize performance (e.g. fixed or variable length)
- The major data types are:
  - text or character
  - number
  - dates and time

Character types (MySQL)

- **CHAR(M):** A fixed-length string that is always right-padded with spaces to the specified length when stored on disc. The range of M is 1 to 255.
- **CHAR:** Synonym for CHAR(1).
- **VARCHAR(M):** A variable-length string. Only the characters inserted are stored – no padding. The range of M is 1 to 65535 characters.
- **BLOB, TEXT:** A binary or text object with a maximum length of 65535 (2^16) bytes (blob) or characters (text). Not stored inline with row data.
- **LONGBLOB, LONGTEXT:** A BLOB or TEXT column with a maximum length of 4,294,967,295 (2^32 - 1) characters.
- **ENUM** (‘value1’,’value2’,…) up to 65,535 members.

Number types (MySQL)

- Integers
  - **TINYINT**: Signed (-128 to 127), Unsigned (0 to 255)
  - **BIT**, **BOOL:** synonyms for TINYINT
  - **SMALLINT**:
    Signed (-32,768 to 32,767), Unsigned (0 to 65,535 – 64k)
  - **MEDIUMINT**:
    Signed (-8388608 to 8388607), Unsigned (0 to 16777215 –16M)
  - **INT / INTEGER**:
    Signed (-2,147,483,648 to 2,147,483,647),
    Unsigned (0 to 4,294,967,295 – 4G or 2^32)
  - **BIGINT**:
    Signed (-9223372036854775808 to 9223372036854775807),
    Unsigned (0 to 18,446,744,073,709,551,615 - 2^64)
  - **Don't** use the "(M)" number for integers

Number types (MySQL)

- Real numbers (fractions)
  - **FLOAT**: single-precision floating point, allowable values: -3.402823466E+38 to -1.175494351E-38, 0, and 1.175494351E-38 to 3.402823466E+38.
  - **DOUBLE / REAL**: double-precision, allowable values: -1.7976931348623157E+308 to -2.2250738585072014E-308, 0, and 2.2250738585072014E-308 to 1.7976931348623157E+308.
  - optional M = number of digits stored, D = number of decimals.
  - Float and Double are often used for scientific data.
  - **DECIMAL**[(M[,D])]: fixed-point type. Good for money values.
  - M = precision (number of digits stored), D = number of decimals

Time types

- **DATE** 1000-01-01 to 9999-12-31
- **TIME** -838:59:59 to 838:59:59
  (time of day or elapsed time)
- **DATETIME** 1000-01-01 00:00:00 to
  9999-12-31 23:59:59
- **TIMESTAMP** 1970-01-01 00:00:00 - ~ 2037
  Stored in UTC, converted to local
- **YEAR** 1901 to 2155

How is DATETIME different to TIMESTAMP?

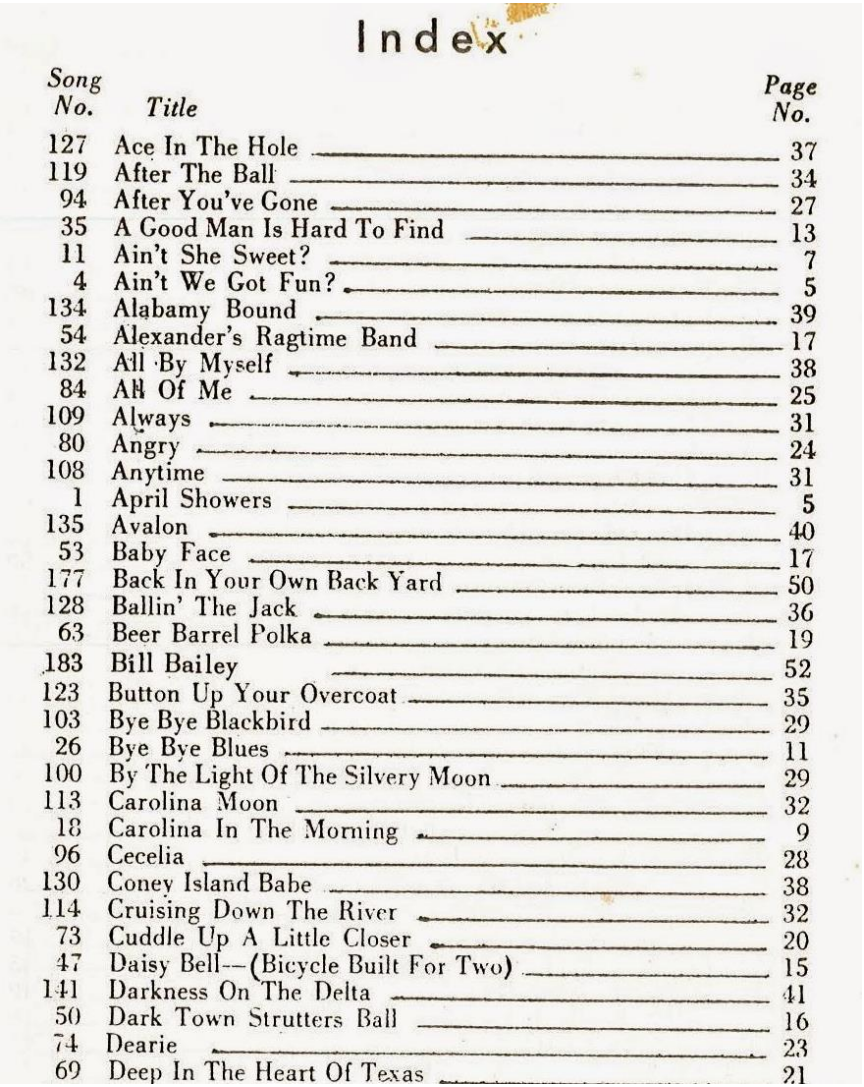
– TIMESTAMP is for automatically adding “now” to a row
– DATETIME is for referring to a time in the real world
– since version 5.6.4, both can store fractions of a second

Indexes

- When running a search query, the bigger the table the more rows that need to be searched. This can get costly, especially for tables with millions or billions of records.
- A database index is a data structure that improves the speed of operations in a table. (MySQL) indexes solve this problem, by taking data from a column in your table and storing it alphabetically in a separate location called an index.

Indexing columns

- Index – a separate file that contains pointers to table rows, to allow fast retrieval
- Similar to an index in a book

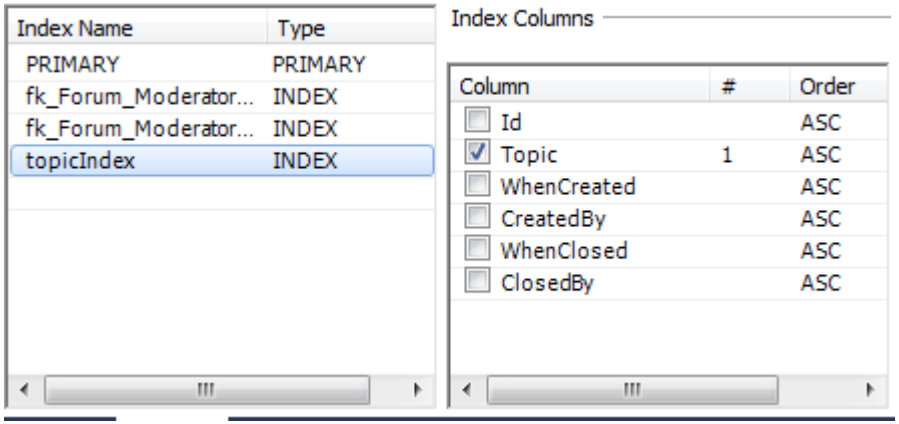


Index

Song No.	Title	Page No.
127	Ace In The Hole	37
119	After The Ball	34
94	After You've Gone	27
35	A Good Man Is Hard To Find	13
11	Ain't She Sweet?	7
4	Ain't We Got Fun?	5
134	Alabamy Bound	39
54	Alexander's Ragtime Band	17
132	All By Myself	38
84	All Of Me	25
109	Always	31
80	Angry	24
108	Anytime	31
1	April Showers	5
135	Avalon	40
53	Baby Face	17
177	Back In Your Own Back Yard	50
128	Ballin' The Jack	36
63	Beer Barrel Polka	19
183	Bill Bailey	52
123	Button Up Your Overcoat	35
103	Bye Bye Blackbird	29
26	Bye Bye Blues	11
100	By The Light Of The Silvery Moon	29
113	Carolina Moon	32
18	Carolina In The Morning	9
96	Cecelia	28
130	Coney Island Babe	38
114	Cruising Down The River	32
73	Cuddle Up A Little Closer	20
47	Daisy Bell--(Bicycle Built For Two)	15
141	Darkness On The Delta	41
50	Dark Town Strutters Ball	16
74	Dearie	23
69	Deep In The Heart Of Texas	21

Indexing columns

- You choose which columns to index
- PKs and FKs are automatically indexed
- Nominate columns to index when creating tables
- PRIMAY | UNIQUE | INDEX | FULLTEXT




```
CREATE TABLE IF NOT EXISTS `UniChat`.`Forum` (
  `Id` INT NOT NULL,
  `Topic` VARCHAR(100) NULL,
  `WhenCreated` TIMESTAMP NULL,
  `CreatedBy` INT NULL,
  `WhenClosed` TIMESTAMP NULL,
  `ClosedBy` INT NULL,
  PRIMARY KEY (`Id`),
  INDEX `fk_Forum_Moderator1_idx` (`CreatedBy` ASC),
  INDEX `fk_Forum_Moderator2_idx` (`ClosedBy` ASC),
  INDEX `topicIndex` (`Topic` ASC),
  CONSTRAINT `fk_Forum_Moderator1`
    FOREIGN KEY (`CreatedBy`)
    REFERENCES `UniChat`.`Lecturer` (`Id`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION,
  CONSTRAINT `fk_Forum_Moderator2`
    FOREIGN KEY (`ClosedBy`)
    REFERENCES `UniChat`.`Lecturer` (`Id`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
ENGINE = InnoDB;
```

When to use Indexes

- use on larger tables to speed up certain types of queries
- on columns which are frequently in WHERE clauses
- or in ORDER BY and GROUP BY clauses
- if column has >100 distinct values but not if <30 values
- So, primary key fields are generally indexed, since they most often feature in the main condition of WHERE clauses.

Id	Title	Classification	ReleaseYear	Genre	CostWholesale	PricePerView
1	Titanic	PG	1999	1	5.02	0.67
2	The Lord of the Ring...	PG	2004	2	1.31	0.58
3	Pirates of the Carib...	PG	2006	2	1.49	1.40
4	Harry Potter and the...	PG	2002	3	3.50	0.75
5	Pirates of the Carib...	PG	2007	2	3.06	1.04
6	Harry Potter and the...	PG	2007	4	4.78	1.47
7	Star Wars: Episode I...	PG	2001	5	4.71	0.72
8	The Lord of the Ring...	PG	2003	3	9.23	0.67
9	Jurassic Park	PG	2001	6	2.02	0.72
10	Harry Potter and the...	PG	2006	4	2.41	1.09
11	Spider-Man 3	PG	2007	2	6.00	0.79

m2 ×

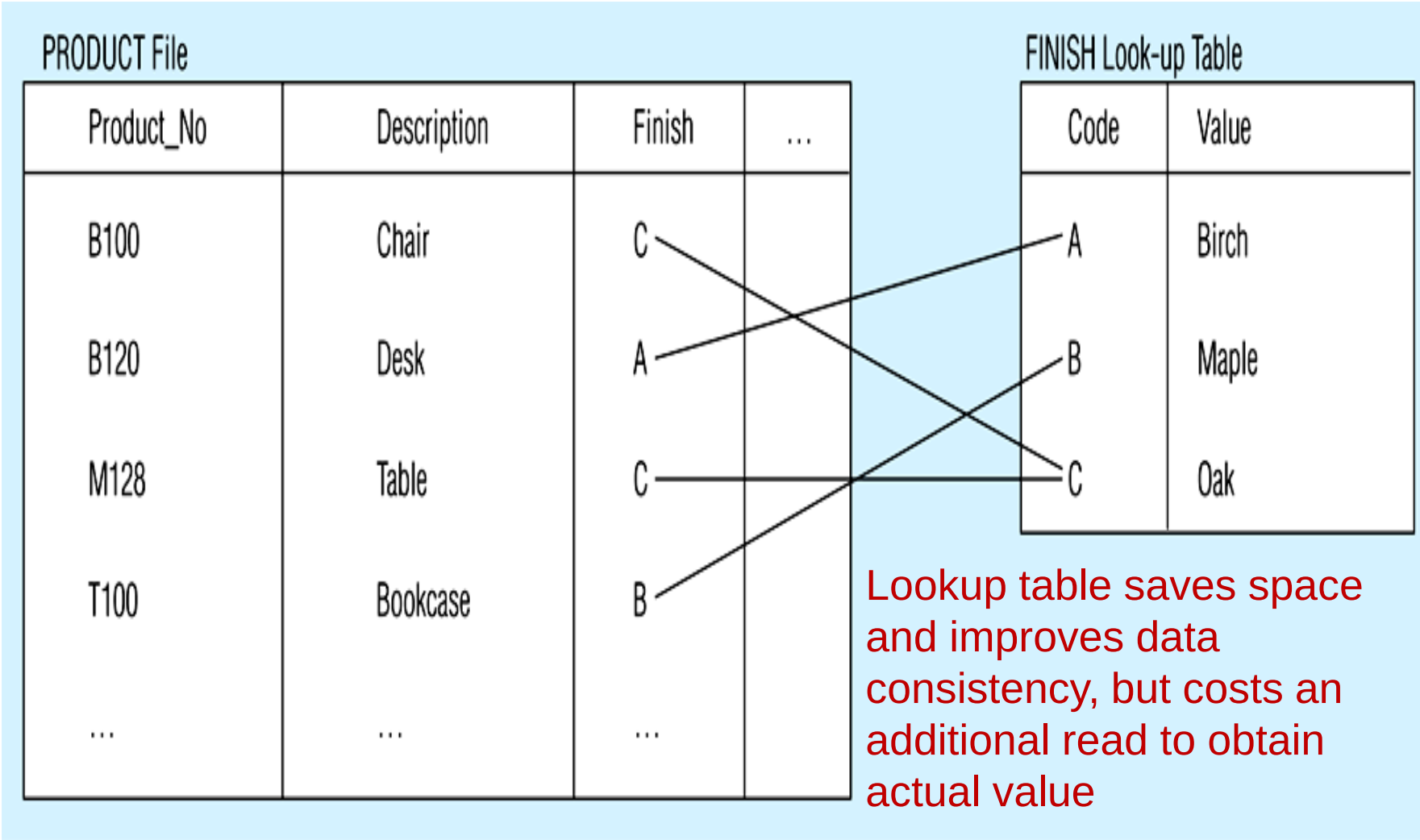
When *NOT* to use Indexes

- Limit the use of indexes for *volatile* databases
  - Why?
  - "Volatile" = data are frequently changed
  - When table data are changed (e.g. by inserts, deletes, updates) indexes need to be updated
  - Indexes slow down inserts and updates, so you want to use them carefully on columns that are FREQUENTLY updated.

De-normalisation

- Normalisation
  - removes data redundancy
  - solves Insert, Update and Delete anomalies
  - makes it easier to maintain information in a consistent state
- However
  - it leads to more tables in the database
  - typically these need to be joined during Selects, which is expensive to do
  - sometimes we decide to ‘de-normalize’

Denormalization example: Look-up table

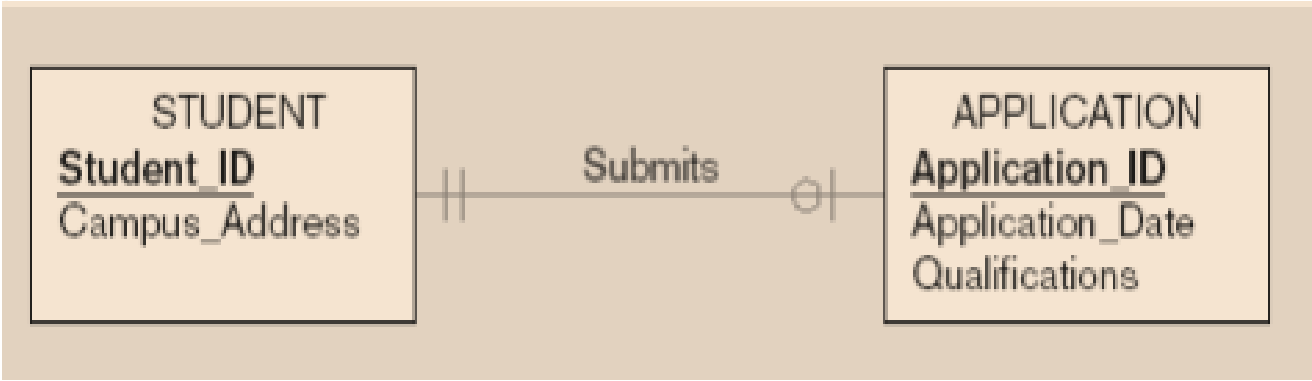


Denormalisation opportunities

- You might want to de-normalise if
  - database speeds are unacceptable
  - there are going to be very few INSERTs, UPDATEs, or DELETEs
  - there are going to be lots of SELECTs that involve the joining of tables

- Examples
  - one-to-one relationship
  - many-to-many relationship with attributes
  - reference data / lookup table
    (1:N relationship where data on 1-side not used in any other relationship)

One-to-One



Normalized relations:

STUDENT

Student_ID	Campus_Address

APPLICATION

Application_ID	Application_Date	Qualifications	Student_ID

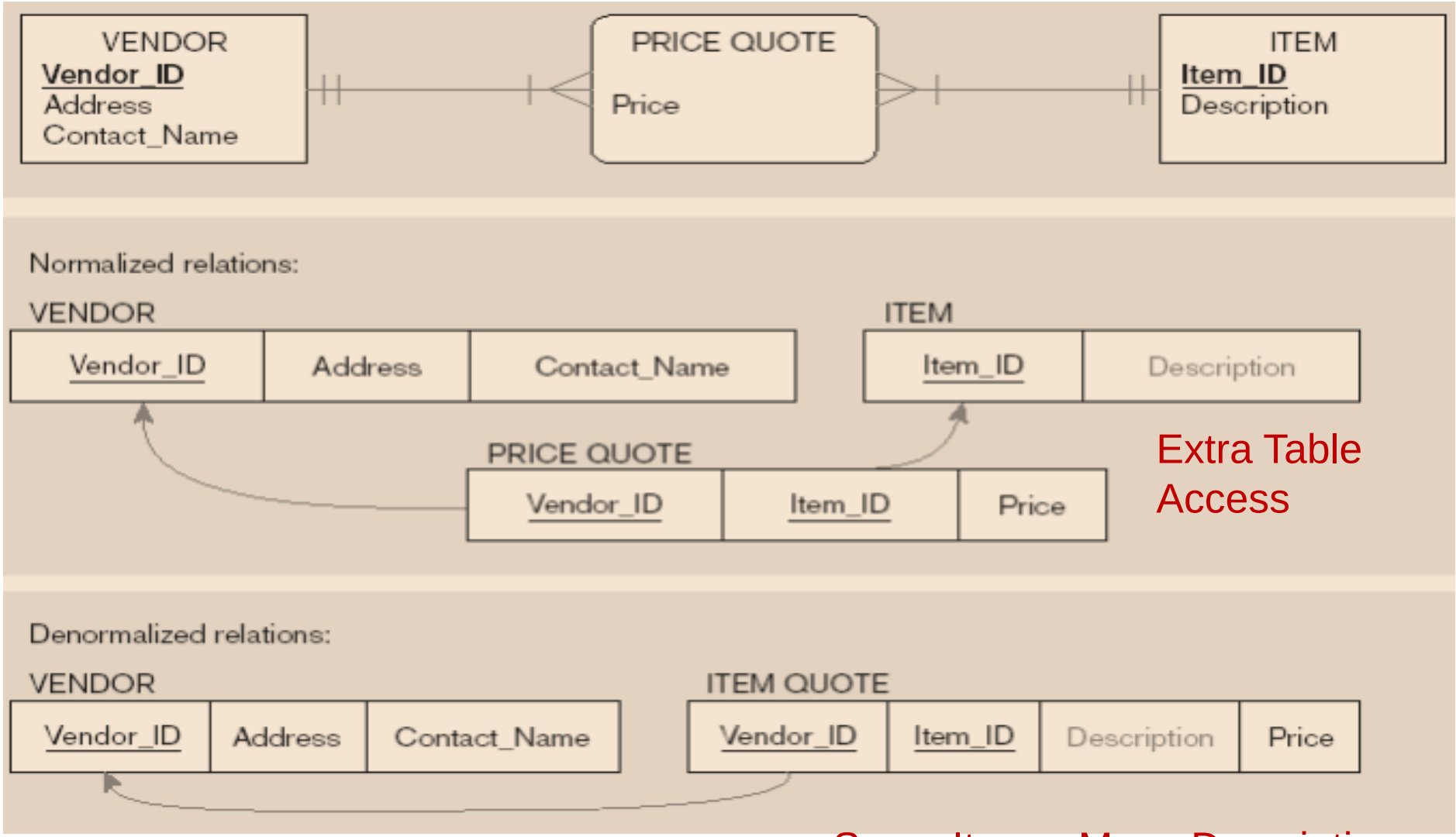
Denormalized relation:

STUDENT

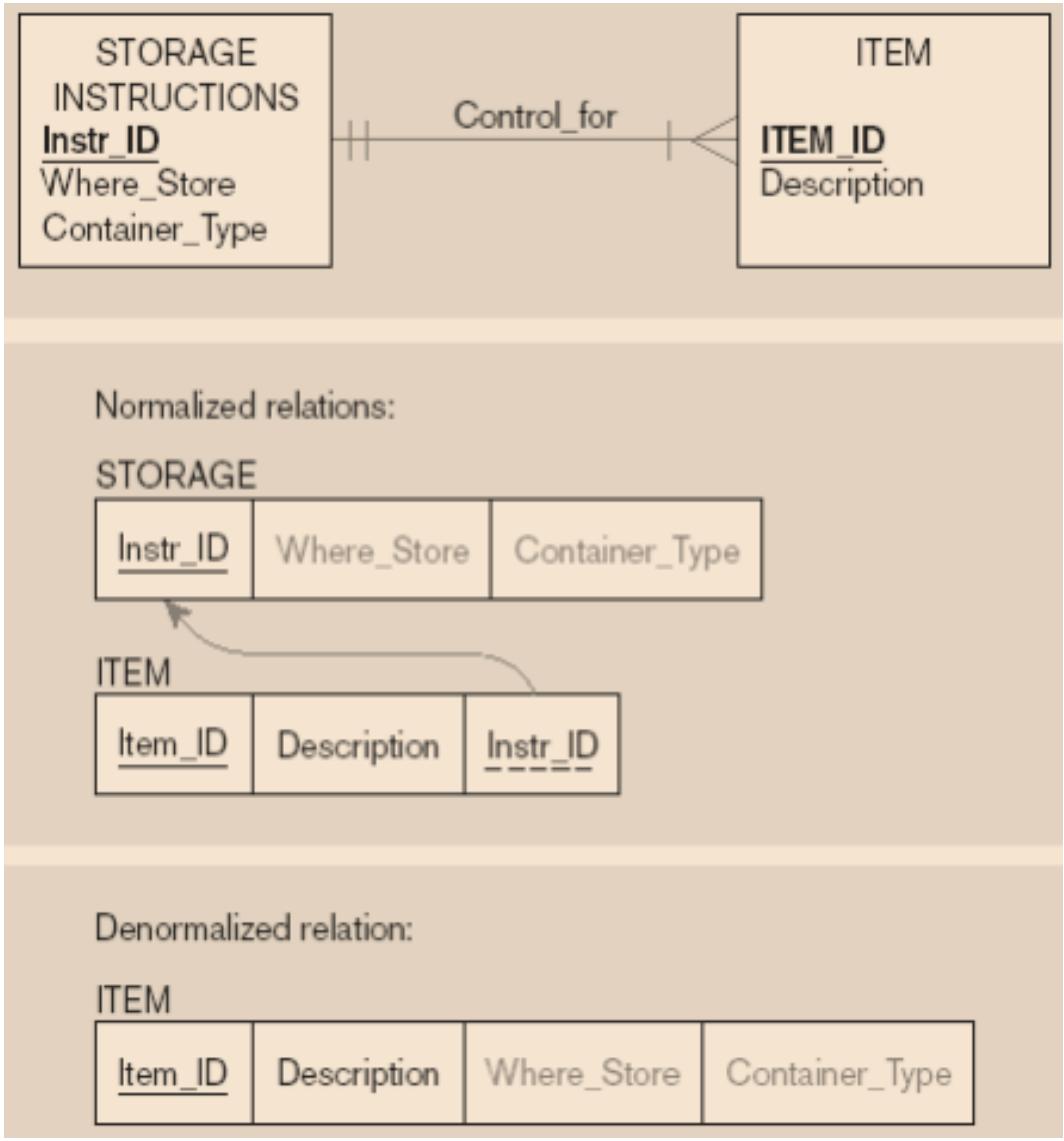
Student_ID	Campus_Address	Application_Date	Qualifications

and Application_Date and Qualifications may be null

Many-to-Many with non-key attributes



Reference Data



extra table access required

data duplication