Avancier Methods (AM)

From logical model to physical database

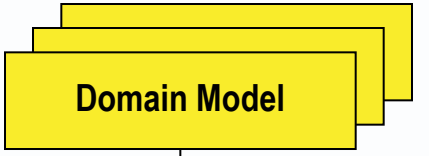
Data structures

Logical domain models

Avancier

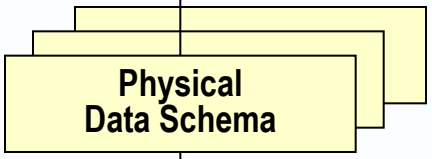
Conceptual

Logical

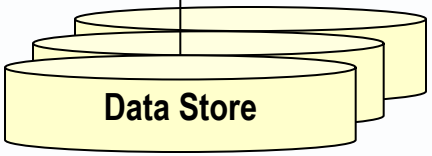


- technology neutral,
- but usually focused on one data store

Physical



Real



Data in Storage

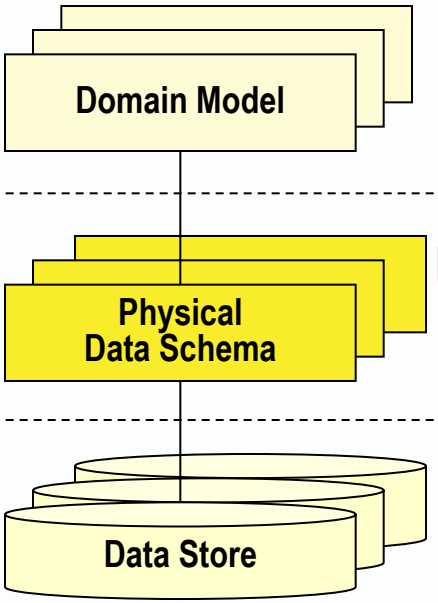
Physical data models

Conceptual

Logical

Physical

Real



A physical data schema or model defines a data structure in a way that is tuned to specific technologies and NFRs

Data in Storage

Physical data models: databases

- The physical data model (though it holds the same data as the logical data model) can be different from the logical one

- Ranging from
  - Slightly denormalised relational database, to
  - Radically restructured non-relational database

Physical database schema types include

- Structured data
  - Relational (RDBMS)
  - Key-value store
  - Column store
  - Document store (e.g. XML or JSON documents)
- Unstructured data (e.g. Hadoop)
  - Data can be anything from free-form to structured
  - A benefit being you can store whatever data is captured, then glue different data sets together later

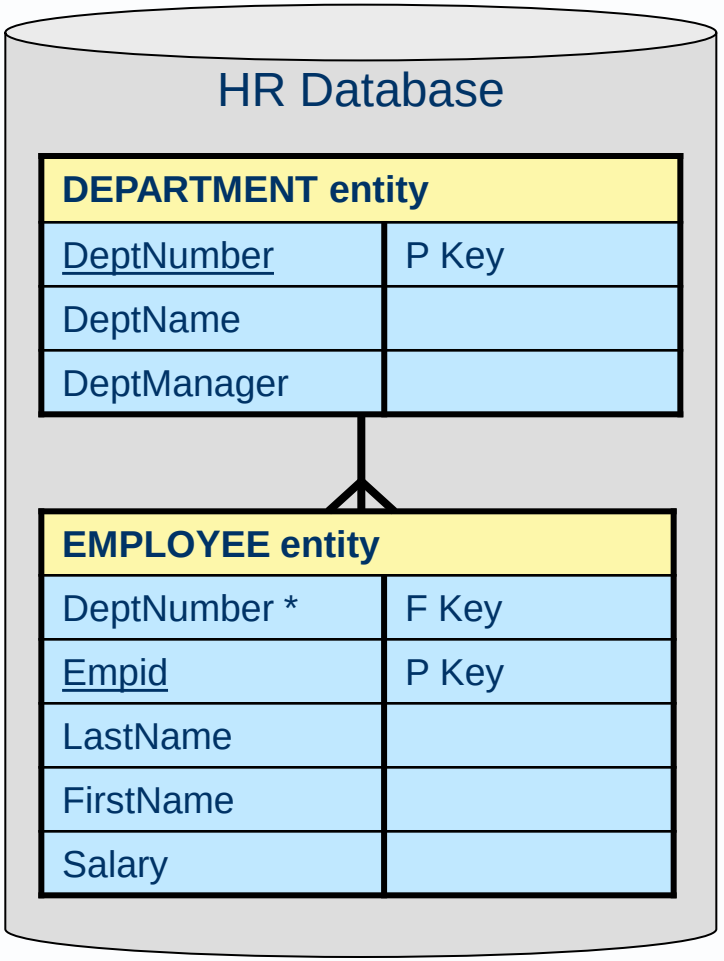
- Non-relational databases are sometimes misleadingly called NoSQL databases.
- They need some kind of SQL if they are to support the coding of queries.

Relational (RDBMS)

- Entities appear in rows of one table
- A typical online transaction updates one row in a table

- An RDBMS suits OLTP-like workloads
- With many discrete transactions

RowId	Deptid	EmpId	Lastname	Firstname	Salary
001	1	10	Smith	Joe	40000
002	7	12	Jones	Mary	50000
003	5	11	Johnson	Cathy	44000
004	3	22	Jones	Bob	55000



RDBMS: Default data architecture solution?

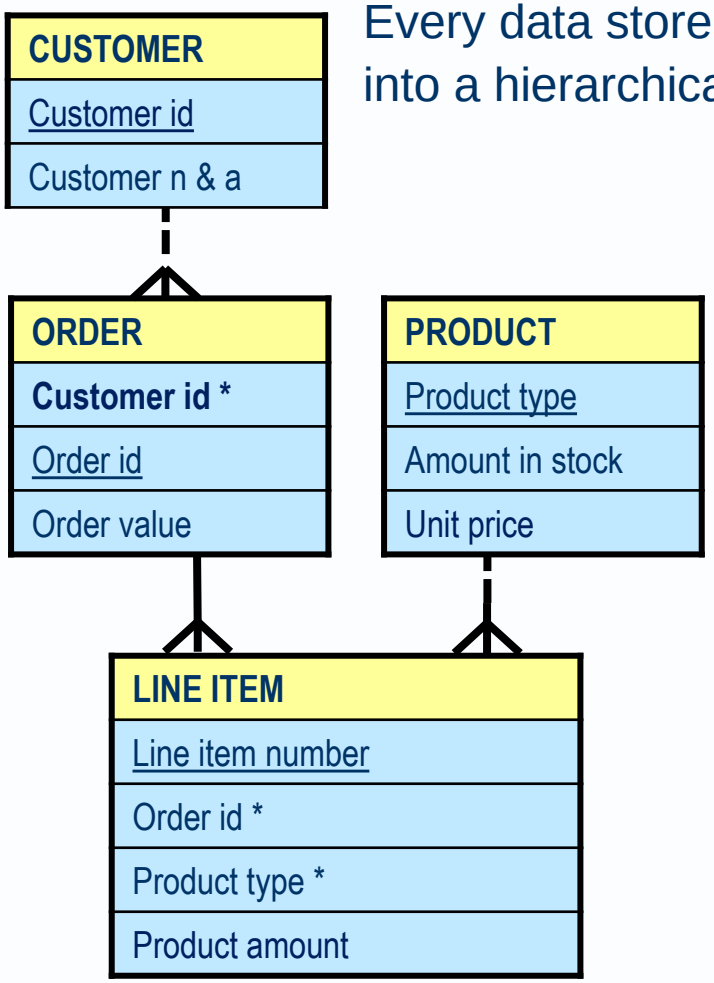
- It is easy and natural to build a relational database to hold the data defined in a logical data model.
- Many enterprise applications have been built on top of a single relational database.
- It is the default data architecture for an enterprise application, and should always be considered.

- You need good reasons to store data in a different way.
- And those reasons are usually non-functional requirements rather than functional ones.

Optimisation the LDM to form a *Physical* Relational Data Model

- Identify processes to be supported, especially
  - batch input and output processes
  - predicted queries and required reports
  - processes that are frequent or have long access paths
- Ensure the data model contains data needed by those processes
- Facilitate process access paths
  - Do access path analysis
  - Add derivable data
  - Add derivable relationships
  - Otherwise denormalise the data structure

Consider how data will be serialised into a required data flow



Every data store can be serialised into a hierarchical / sequential data flow

Serialisation

Customer Order History
- **Customer id**
- **Customer name and address**
- Orders Placed
  - **Order id**
  - **Order value**
  - Products Ordered
    - **Product type**
    - **Product amount**
  - Products Ordered End
- Order Placed End

Customer Order History END

Serialisation

Product Demand
- **Product type**
- **Amount on hand**
- Products ordered
  - **Product amount**
  - **Order id**
- Products Ordered End

Product Demand End

Don’t forget the numbers: international car parts manufacturer

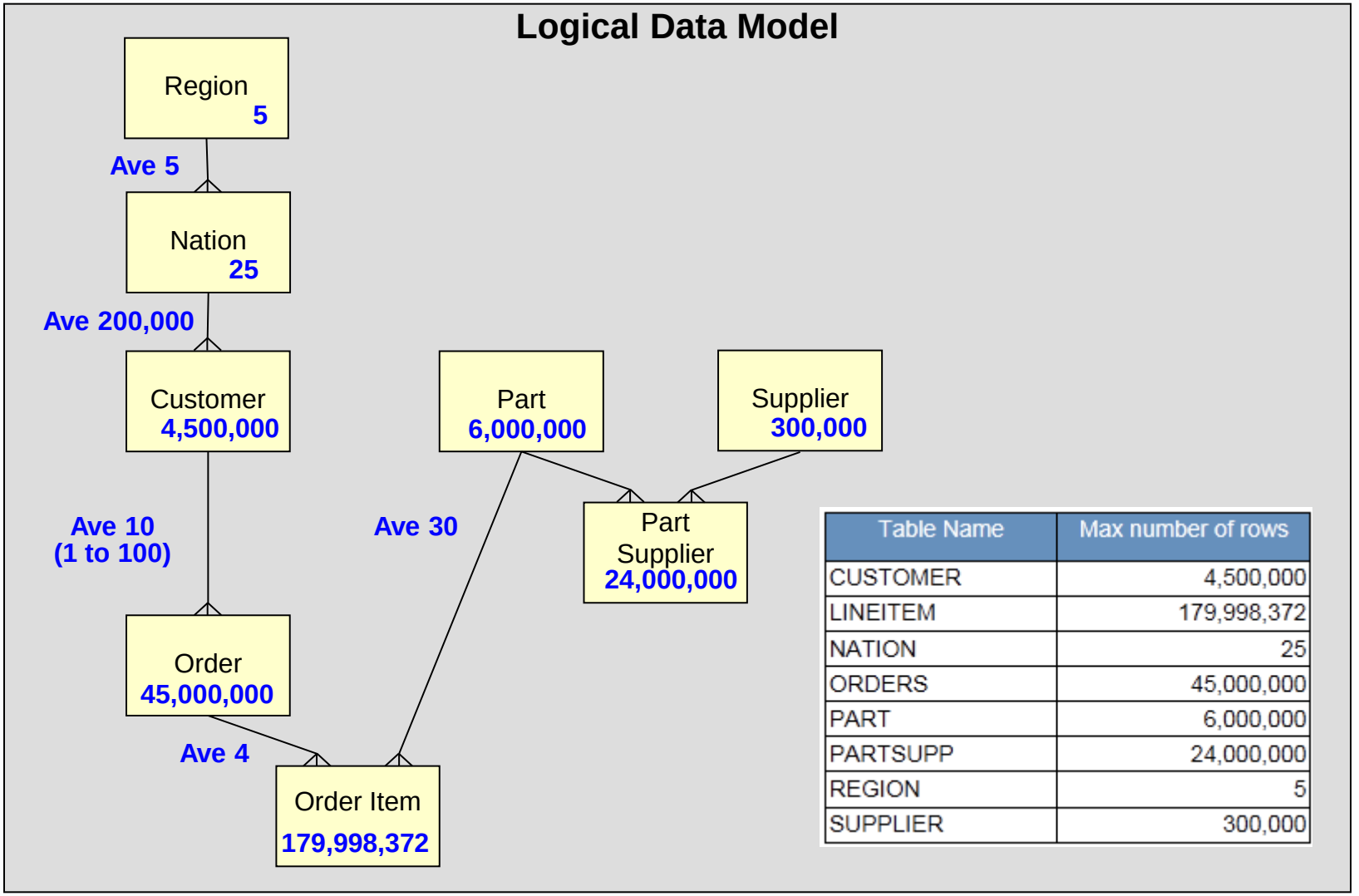
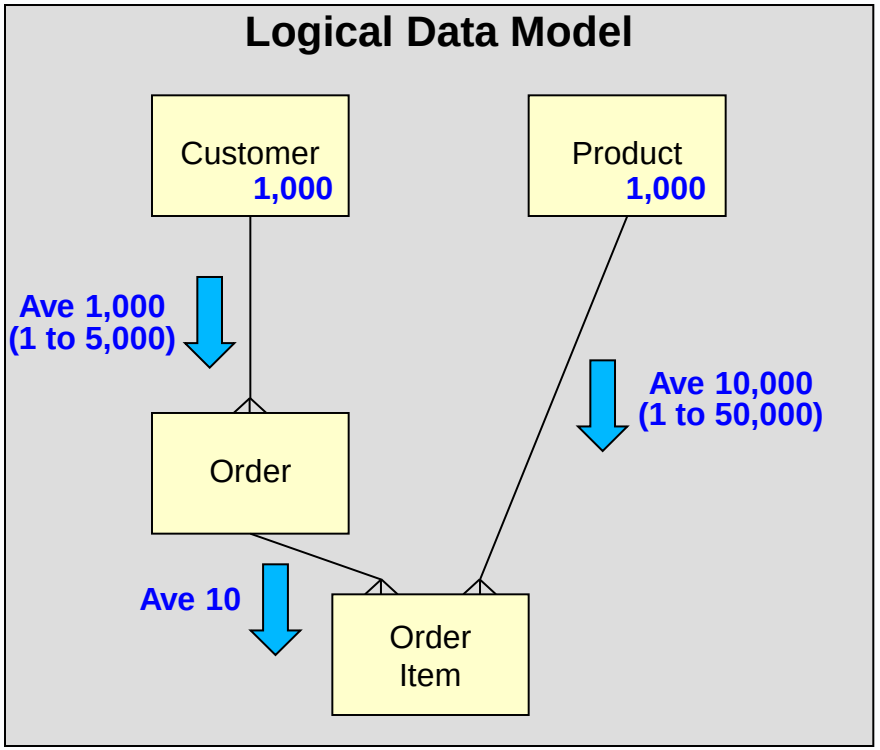


Table Name	Max number of rows
CUSTOMER	4,500,000
LINEITEM	179,998,372
NATION	25
ORDERS	45,000,000
PART	6,000,000
PARTSUPP	24,000,000
REGION	5
SUPPLIER	300,000

Don’t forget the numbers: school stationary supplies

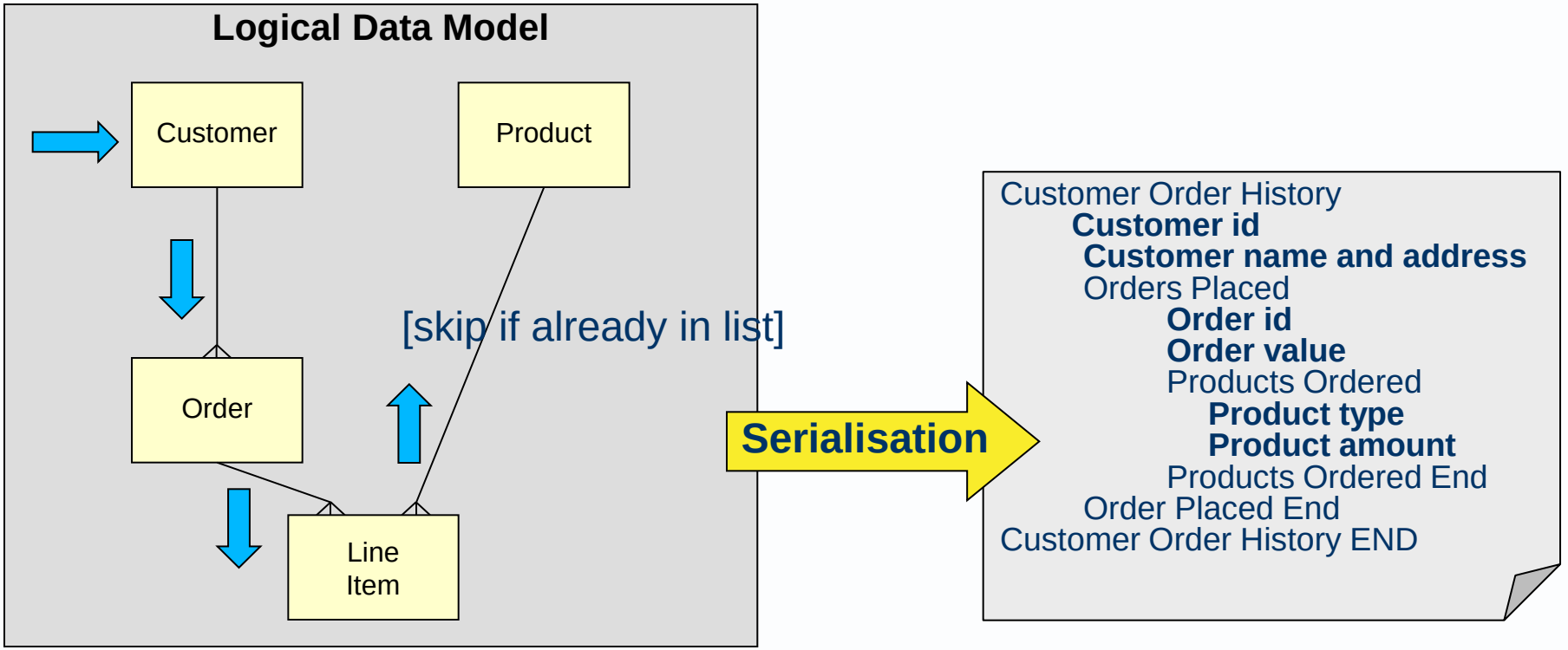
- Given a Logical Data Model, define
  - the volumes of kernel entities,
  - the population of each relationship,
  - expected growth rates.



The numbers influence the physical design

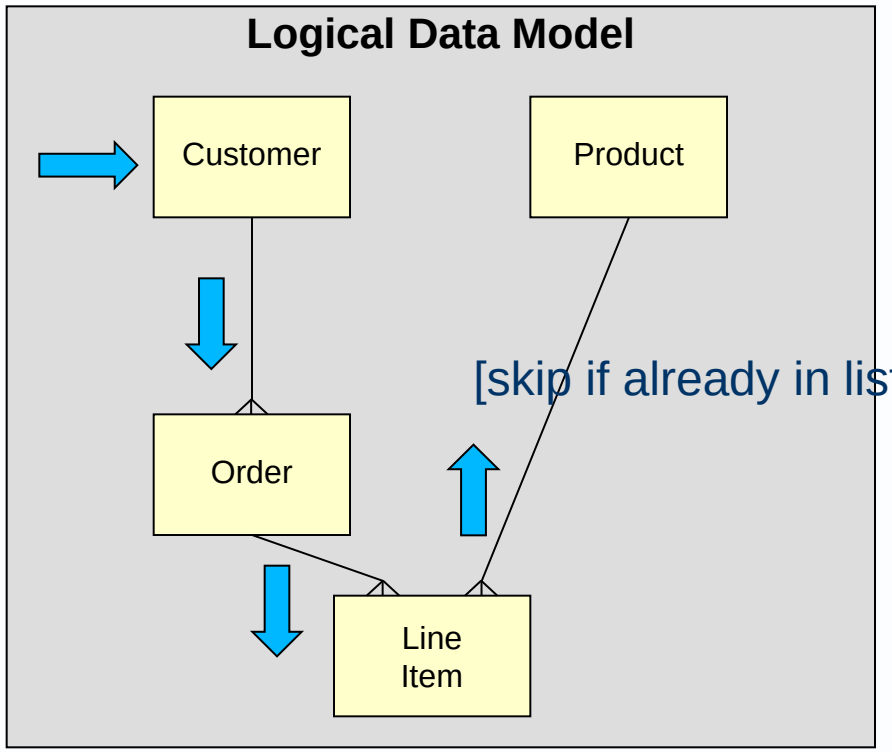
Do access path analysis – is it feasible?

- "List all the products ordered by a customer"
- First, is it feasible? YES

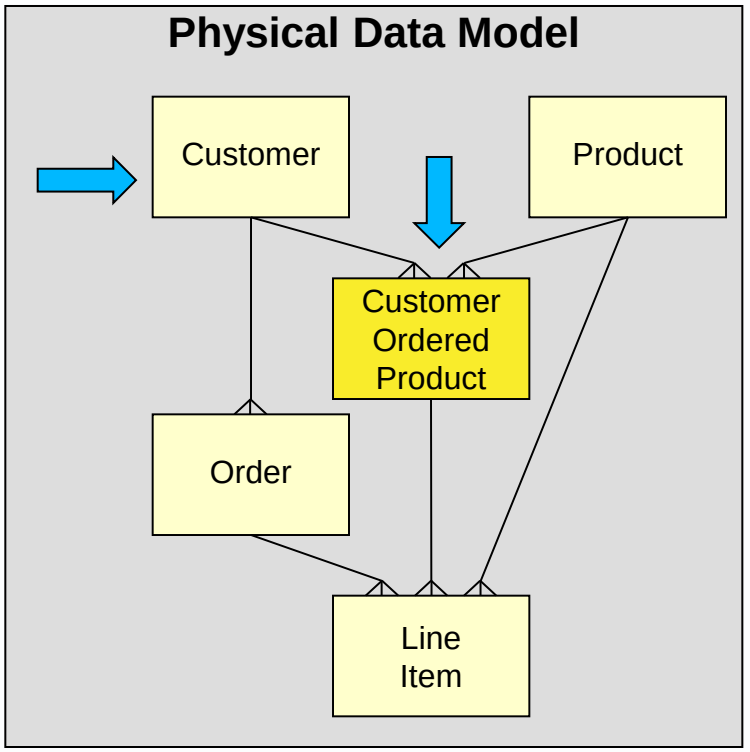


Do access path analysis – is it fast enough?

- The typical customer (a school) has placed 1,000 orders, each with 10 items, for our stationery products
- But each as ordered only 15 products

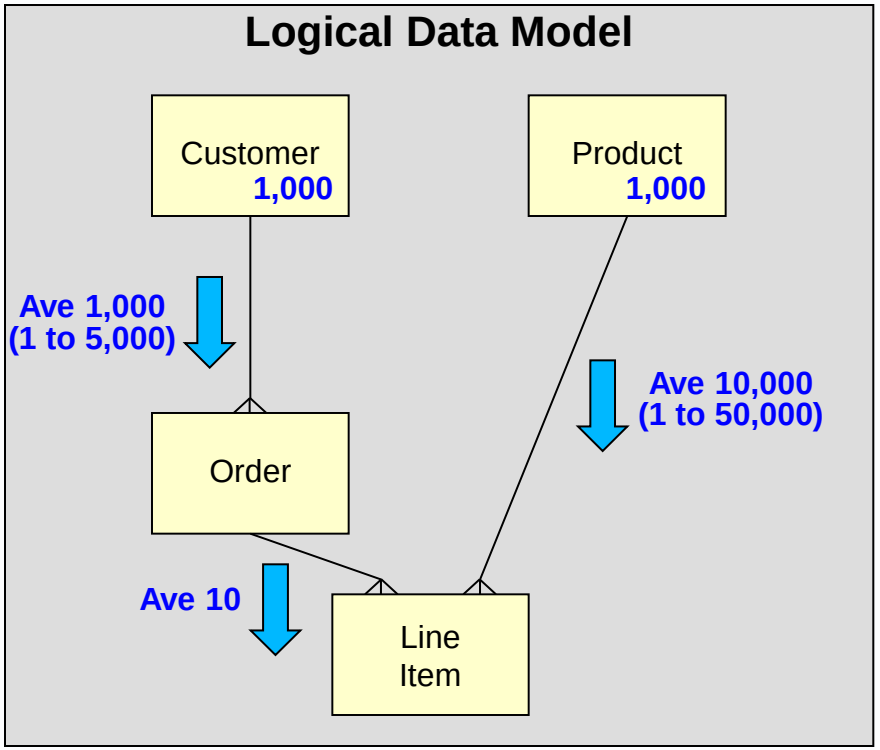


- You can remove redundant accesses by adding redundant data or redundant relationships



Don’t forget the numbers

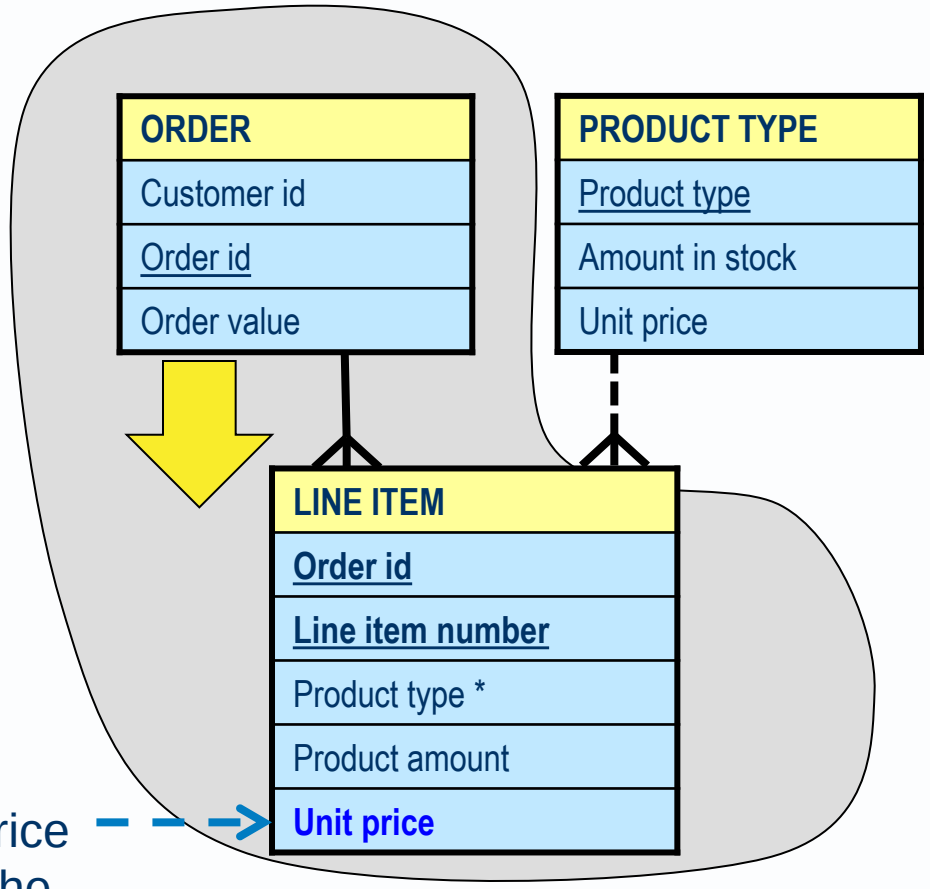
- Given a Logical Data Model, define
  - the volumes of kernel entities,
  - the population of each relationship,
  - expected growth rates.



The numbers influence the physical design

Clustering related data on the same block/page

- An old-fashioned optimisation technique
- Cluster detail entities with *one* of several parents
- Rule of thumb: “the least dependent occurrence rule”
- An Order has few Order Lines
- A Product has many
- So cluster the Order Lines on the same block/page as the Order



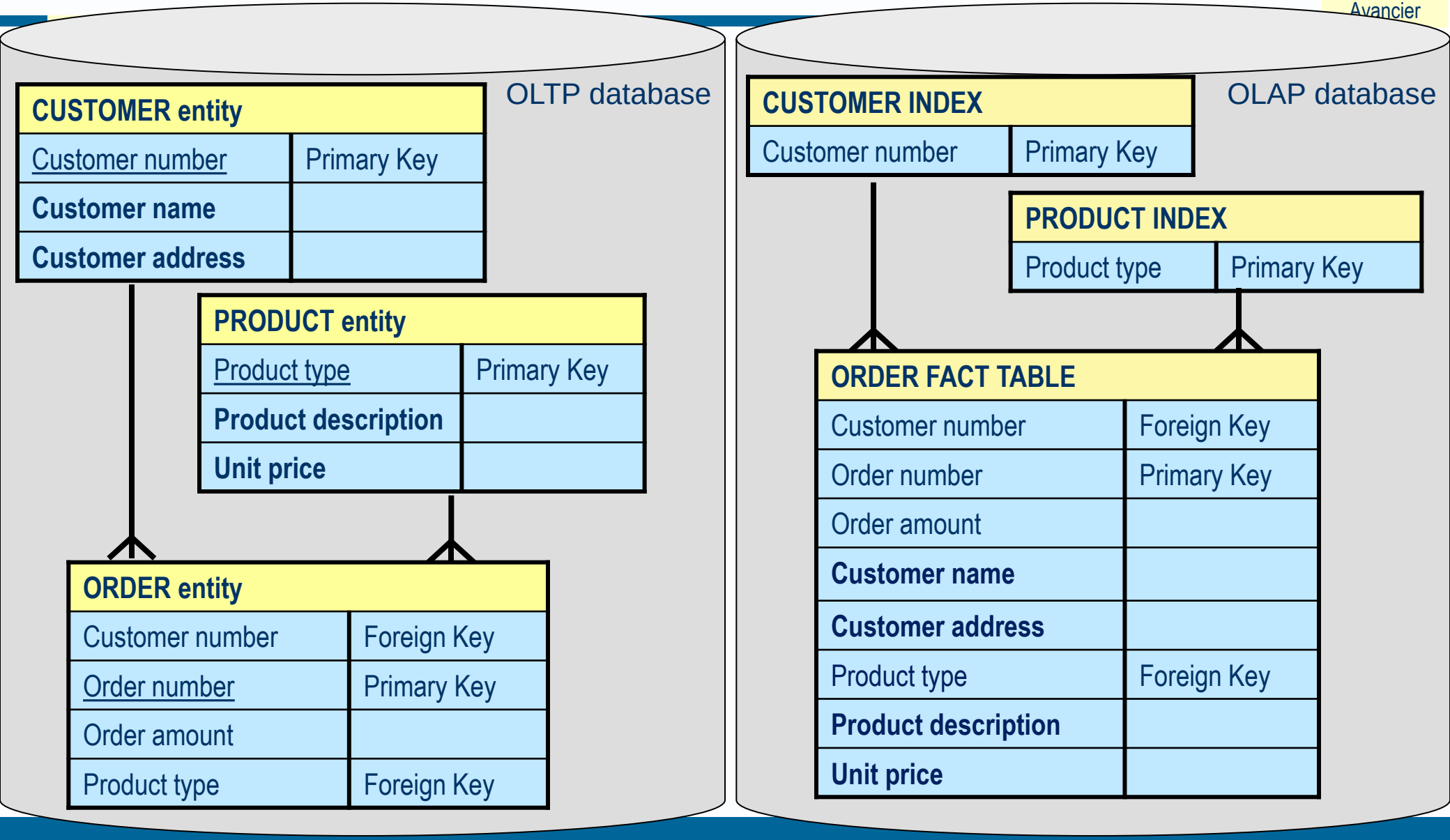
- Copying the price stops the price change transaction affecting the aggregate, which is probably the business rule anyway

Separation of update and query data stores

- You can remove redundant accesses by adding redundant data or redundant relationships.
- This includes adding indexes.

- Where adding redundant data and relations is not enough, then you can separate data stores for updates and queries.
- This is a common and powerful technique, since you can de-normalise the query database.

Denormalisation for faster enquiry/report processes (naive picture)



Queries that take a long time

- One attribute value may appear in only a few of many rows
- Q) List all employees with Lastname = Jones

RowId	Deptid	EmpId	Lastname	Firstname	Salary
001	1	10	Smith	Joe	40000
002	7	12	**Jones**	Mary	50000
003	5	11	Johnson	Cathy	44000
004	3	22	**Jones**	Bob	55000

- The query must look at every row in the whole table unless
  - a) we introduce an index with an entry for every attribute value
  - b) we use a data store designed to support such a query

Alternatively

- Where de-normalising the query database is still not enough
  - meaning some queries still take too long or hog the resources.

- Then look at non-relational databases
  - that restructure the logical data model in a more radical way.

Column store

- Attribute values appears as rows.
- Rows appear as attributes

Empid	Rows
10	001
12	002
11	003
22	004

Lastname	Rows
Smith	001
Jones	**002, 004**
Johnson	003

Firstname	Rows
Joe	001
Mary	002
Cathy	003
Bob	004

Salary	Rows
40000	001
5000	002
44000	003
55000	004

- Column Store
  - Suits OLAP-like workloads (e.g. data warehouses)
  - A smaller number of complex queries over lots of data
  - Efficient where columns have many duplicate values (including null values)

Column Store

- One application loads one million rail tickets every night into an RDBMS, for apportionment of money between stakeholders

Ticketid	From	To	Date	Time	Seat
001	Lond	Newc	30122030	10.30	A36
002	Newc	Edin	30122030	10.30	A36
003	Edin.	Newc	30122030	18.30	A36
004	Newc	Lond	30122030		

- Another application loads those tickets into a data warehouse for analysis.
- Much duplication of attribute values in different tickets/rows.

From	Tickets
Lond	001
Newc	002, 004
Edin	003

To	Tickets
Lond	004
Newc	001, 003
Edin	002

Date	Tickets
30122030	001, 002,003, 004

Time	Tickets
10.30	001, 002
18.30	003

Seat	Keys
A36	001, 002, 003,

Key Value database

- Like one giant database table, in which every attribute value is a distinct row.
- Scalable to handle millions of queries per second by adding more servers.

Key	Value
001_empid	10
003_lastname	Johnson
002_empid	12
003_empid	11
001_firstname	Joe
004_empid	22
001_lastname	Smith
002_lastname	Jones
002_salary	5000
002_lastname	Jones
003_firstname	Cathy
004_firstname	Bob
001_salary	40000
002_firstname	Mary
003_salary	44000
004_salary	55000

Document store

- No pre-defined schema: documents are self-describing
  - <orderhistory>
    - <firstname>Bob</firstname>
    - <lastname>Smith</lastname>
    - <email type=Home>bob.smith@gmaile.com</email>
    - <phone type=Cell>(123) 456-7890</phone>
    - <phone type=Work>(890) 765-4321</phone>
    - <order>
      - <ordernumber>123.</ordernumber>
      - <orderamount>200</orderamount>
      - <productnumber>4444</productnumber>
      - <productdescription>A4 ream</productdescription>
    - </order>
    - <order>
      - <ordernumber>124.</ordernumber>
      - <orderamount>70</orderamount
      - ….
      - ….

Order History SEQUENCE
- **Firstname**
- **Lastname**
- **Email addresses**
- **Phone numbers**
- Set of Orders ITERATION
  - Order SEQUENCE
    - **Order number**
    - **Order amount**
    - **Product type**
    - **Product description**
    - Order END
- Set of Orders END

Order History END

A history of databases

- The history of databases is much about making it easier to
  - store data that is captured via data entry forms and transactions
  - retrieve the data that is wanted
  - store increasing volumes of data
  - ensure data quality and consistency

- Some of what follows is edited from “The history of data processing” http://www.infoq.com/presentations/db-history-data-processing

Databases since 1960

- Multi-Value, Hierarchical
  - PICK, IMS, IDS, ADABAS
- Relational
  - CODASYL, System R (SEQUEL), INGRES (QUEL), Mimer, Oracle
- RBMDS with SQL STANDARD
  - DB2, Teradata, Informix, Sybase, Postgres
- OODBMS, ORDBMS
  - Versant, Objectivity, Gemstone, Informix*, Oracle*
- MPP Query and NoSQL
  - Netezza, Paraccel, Vertica, MongoDB, CouchBase, Riak, Cassandra
- NewSQL
  - SciDB, NuoDB, JethroDB, Metanaulx
- Future?
  - Spanner, F1

Storing BIG data

- **Horizontal partitioning**
  - Divide the rows of a table into several tables and store them.
  - For example, 'User Table' with identical schema can be divided into
    - 'User Table #0' in which users less than 13 years are stored
    - 'User Table #1' in which users 13 or greater than 13 years old.
- **Database sharding**
  - Stores horizontal partitions in physically separate databases.
  - For example:
    - Store users less than 13 years old in database 0
    - Store users 13 or greater than 13 years old in database 1
- Done for performance and scalability
- Increases system complexity

Increasing complexity

- Remember, anything not done by the DBMS becomes a programmer's task.
- An RDBMS is likely to provide all these services:
  - Standard API/Query Layer
  - Transactions/consistency
  - Query optimisation
  - Data navigation, joins
  - Data access *
  - Storage management *

- * A NoSQL database may give you only the last two.

- "Eventual consistency" is a nice way of saying "not correct".
- "Transaction malleability" is a nice way of saying "broken".

When to use “NoSQL” databases?

- Is your data really just a collection forms, each accessible using one key
  - Document store
- Is your data really just a giant hash lookup?
  - Key-value store
- Do you need full-text searching?
  - Text-indexing engine
- Will you get questions about your data that you can't predict?
  - Then better make sure your data **also** ends up in an RDBMS.

KISS

- If you are storing structured business data
- Start with understanding the logical data model
- Look at whatever database best suits your data entry and retrieval/reporting requirements
- You might end up needing an RDBMS for
  - Consistency
  - Unpredictable queries