Designing for document databases

Excerpts from

"NoSQL for Mere Mortals"

Dan Sullivan

Introduction

- Designers have many options when it comes to designing document databases.
- The flexible structure of JSON and XML (among others) documents is a key factor in this → flexibility.
- A designer can embed lists within lists within a document.
- Another designer can create separate collections to separate types of data.

This freedom should not be construed to mean all data models are equally good—**they are not**!

- Relational database designers can apply rules of normalization

to help them assess data models.

- A typical relational data model is designed to avoid **certain** data anomalies when inserts, updates, or deletes are performed.

- For example, if a database maintained multiple copies of a customer's current address, it is possible that one or more of those addresses are updated but others are not.
- In that case, which of the addresses is actually the current one?

- In another case, if you do not store customer information (personal data) separately from the customer's orders, then all personal data of the customer could be deleted if all his orders are deleted…

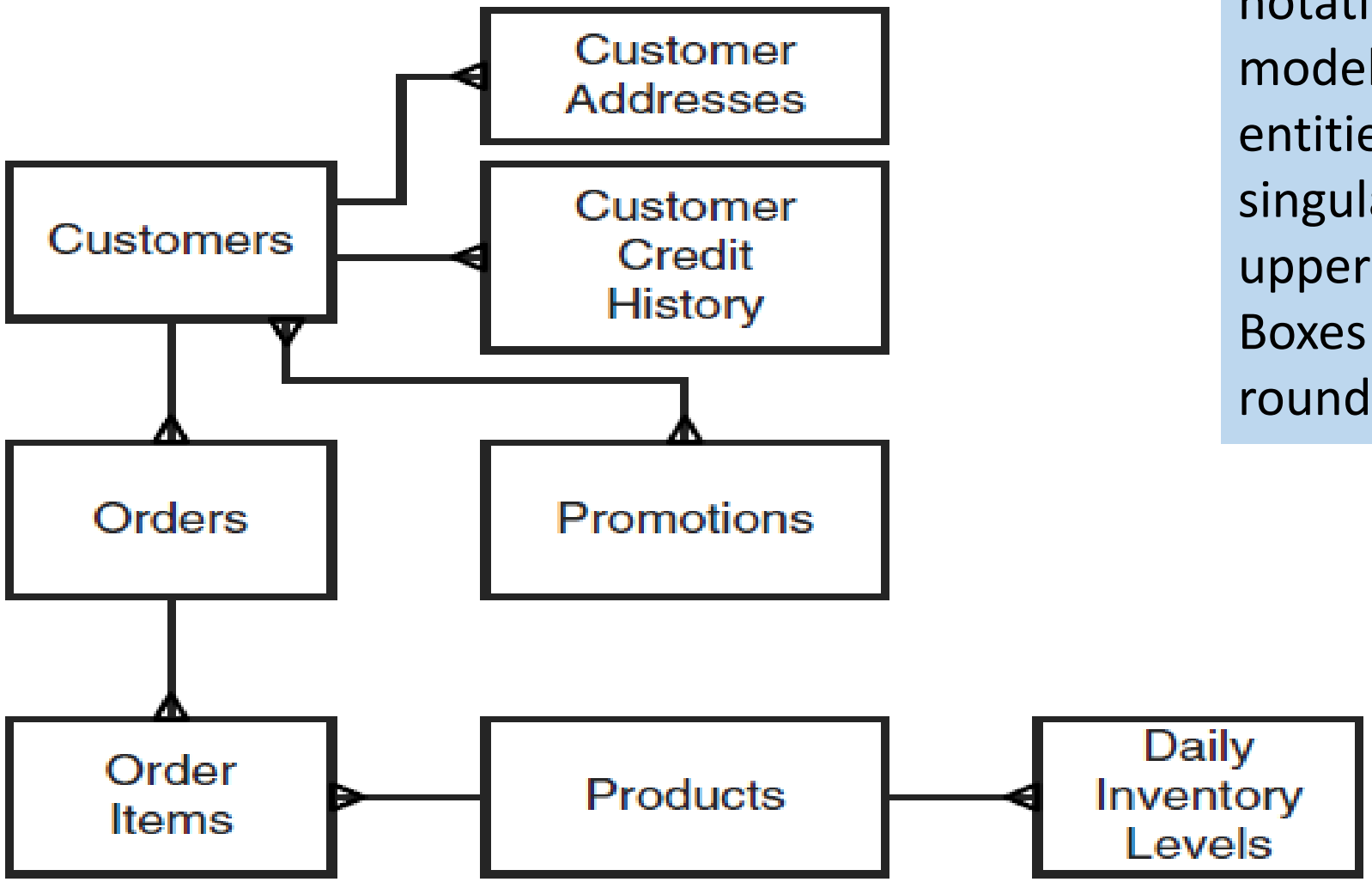
- Document database modelers depend more on heuristics, or rules of thumb*, when designing databases.
- **You must consider how users will query the database, how much inserting will be done, and how often and in what ways** documents **will be updated.**
- The rules are not formal**, they are not logical rules like normalization rules.

**A delicate and undesirable situation!
☹

*Regla práctica

Normalization, Denormalization, and the Search for Proper Balance

- Unless you have worked with relational databases, you probably would not guess that normalization has to do with eliminating redundancy (some type of redundancy).
- Redundant data is considered a bad (or at least undesirable) thing in the theory of relational database design.
- Redundant data is the root of some anomalies, such as two current addresses when only one is allowed.
- Consider the following model



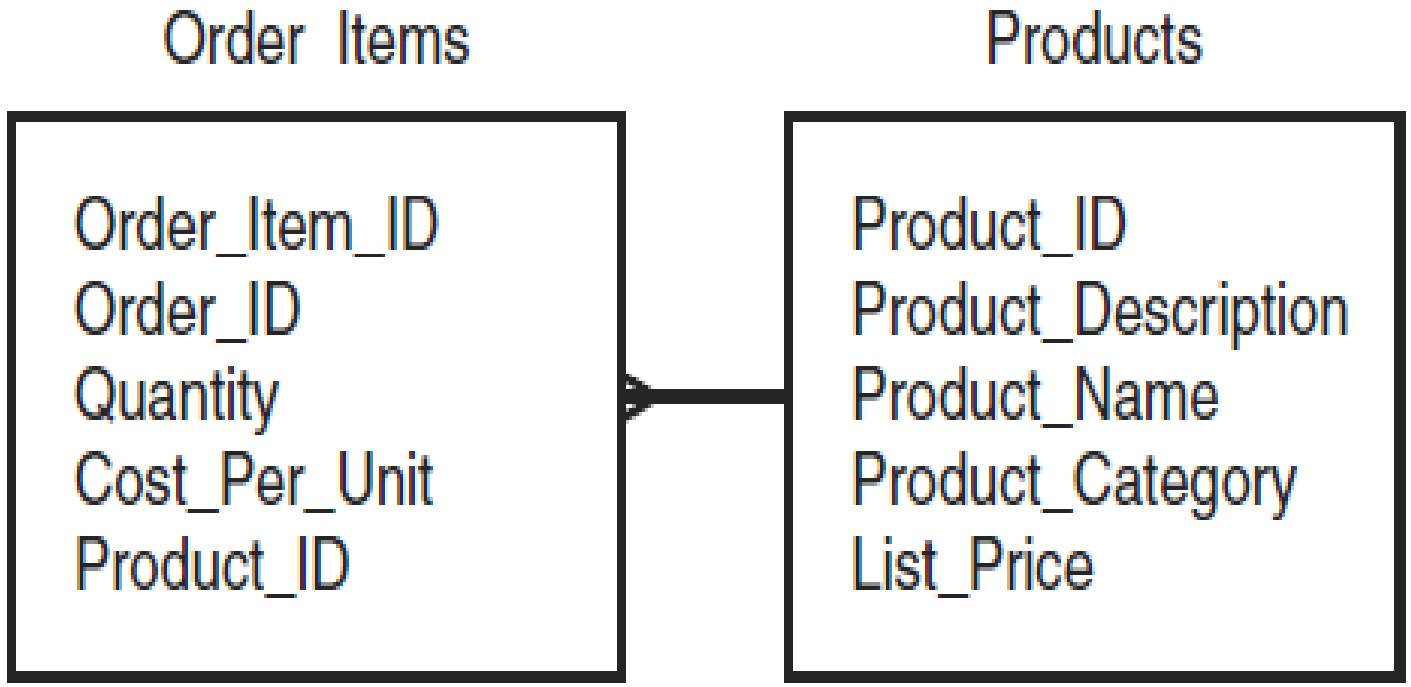
Note: This is not Richard Barker's notation for E-R model: names of entities should be in singular and in uppercase.
Boxes must be rounded

- This figure depicts a simple normalized model.
- Even this simple model requires (traditionally) nine tables to capture a basic set of data about the entities:

customer, order, order_item, customeraddress, customercredhry, promotion, product, dailyinventory, **custxprom** (the last one resulting from the many-to-many relationship between Customers and Promotions).

The Need for Joins

- Developers of applications using relational databases often have to work with data from multiple tables.
- Consider the Order Items and Products entities shown in the next figure.



Note: Again, the author is not following Barker's notation: for example, Product_ID should not be in Order Items…

- If you were designing a report that lists an order with all the items on the order, you would probably need to include attributes such as the name of the product, the cost per unit, and the quantity.
- The name of the product is in the Products table, and the other two attributes are in the Order Items table (see next figure).

Order Items				
Order_Item_ID	Order_ID	Quantity	Cost_Per_Unit	Product_ID
1298	789	1	$25.99	345
1299	789	2	$20.00	372
1300	790	1	$12.50	591
1301	790	1	$20.00	372
1302	790	3	$12.99	413

Products				
Product_ID	Product_Description	Product_Name	Product_Category	List_Price
345	Easy clean tablet cover that fits most 10" Android tablets.	Easy Clean Cover	Electronic Accessories	25.99
372	Lightweight blue ear buds with comfort fit.	Acme Ear Buds	Electronic Accessories	20
413	Set of 10 dry erase markers.	10-Pack Markers	Office Supplies	15
420	60"×48" whiteboard with marker and eraser holder.	Large Whiteboard	Office Supplies	56.99
591	Pack of 100 individually wrapped screen wipes.	Screen Clean Wipes	Office Supplies	12.99

- In relational databases, modelers often start with designs like the one we saw earlier in the first figure.
- Normalized models such as this minimize redundant data and avoid the potential for data anomalies.

- Document database designers; however, often try to store related data together in the same document.
- This would be equivalent to storing related data in one table of a relational database.
- You might wonder why data modelers choose different approaches to their design. It has to do with the tradeoffs between performance and potential data anomalies.

- To understand why normalizing data models can adversely affect performance, let us look at an example with multiple joins.
- Note that from a theoretical point of view this is a problem of the products (DBMS) **not an inherent** problem of the normalized relational model.

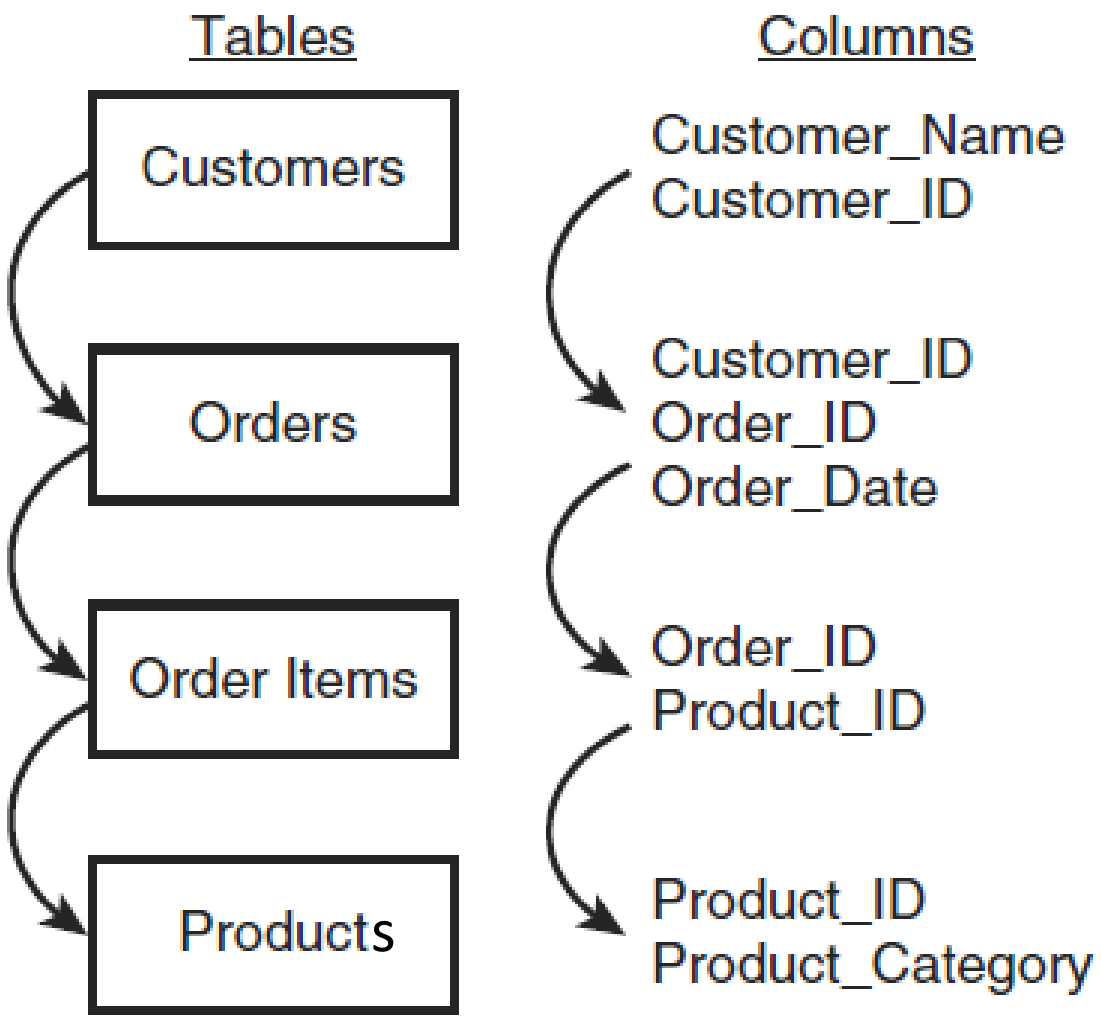
Executing Joins: The Heavy Lifting of Relational Databases

- Imagine you are an analyst and you have decided to develop a promotion for customers who have bought electronic accessories in the past 12 months.
- The first thing you want to do is understand who those customers are, where they live, and how often they buy from your business.
- You can do this by querying the Customer table.

Heavy litfting = Serious or difficult work

- You do not want all customers, though—just those who have bought electronic accessories.
- That information is not stored in the Customer table, so you look to the Orders table. The Orders table has some information you need, such as the date of purchase.
- This enables you to filter for only orders made in the past 12 months.

- The Orders table; however, does not have information on electronic accessories, so you look to the Order Items table.
- This does not have the information you are looking for, so you turn to the Products table.
- The Products table has a column called Product_Category, which indicates if a product is an electronic accessory. You can use this column to filter for electronic accessory items.
- At this point, you have all the data you need.


Tables
Columns
Customers
Orders
Order Items
Products
Customer_Name
Customer_ID
Customer_ID
Order_ID
Order_Date
Order_ID
Product_ID
Product_ID
Product_Category

- To get a sense of how much work is involved in joining tables, let us consider pseudocode for printing the name of customers who have purchased electronic accessories in the last 12 months:

```
for cust in get_customers():
   for order in get_customer_orders(cust.customer_id):
      if today() - 365 <= order.order_date:
         for order_item in get_order_items
           (order.order_id):
          if 'electronic accessories' =
              get_product_category(order_item.product_id):
                 customer_set = add_item
                   (customer_set,cust.name);

for customer_name in customer_set:
   print customer_name;
```

- Let us assume there are 10000 customers in the database.
- The first for loop will execute 10000 times. Each time it executes, it will look up all orders for the customer.
- If each of the 10000 customers has, on average, 10 orders, then the for order loop will execute 100000 times.
- Each time it executes, it will check the order date.

- Let us say there are 20000 orders that have been placed in the last year.
- The for order _ item loop will execute 20000 times. It will perform a check and add a customer name to a set of customer names if at least one of the order items was an electronic accessory.

- However, looping through rows of tables and looking for matches is one—rather **inefficient—way** of performing joins.
- The performance of this join could be improved. For example, indexes could be used to more quickly find all orders placed within the last year. Similarly, indexes could be used to find the products that are in the electronic accessory category.

- Databases implement query optimizers to come up with the best* way of fetching and joining data.
- In addition to using indexes to narrow down the number of rows they have to work with, they may use other techniques to match rows: they could, for example, calculate hash values of foreign keys to quickly determine which rows have matching values.

* It is not always possible to find the “best” way…

Database researchers and vendors have made advances in query optimization techniques, but executing joins on large data sets can still be time consuming and resource intensive.

What Would a Document Database Modeler Do?

- Document data modelers have a different approach to data modeling than most relational database modelers.
- Document database modelers are probably using a document database for its scalability, its flexibility, or both.
- For those using document databases, avoiding data anomalies is still important, but they are willing to assume more responsibility* to prevent them in return for scalability and flexibility.

* A very dangerous situation

- So how do document data modelers and application developers get better performance?
- They minimize the need for joins.
- This process is known as denormalization*.
- The basic idea is that data models should store data that is used together in a single data structure, such as a table in a relational database or a document in a document database.

* Some authors use the term **denormalization** in a more general way, for example, when adding redundant attributes to a relation (derived attributes). See for example: "**When and How You Should Denormalize a Relational Database**" https://rubygarage.org/blog/database-denormalization-with-examples

Is this idea a novelty? No!

- Let us see C. J. Date in his book “Introducción a los sistemas de bases de datos” 1993!:

3. Una vez más, hacemos hincapié en que los lineamientos de normalización son *sólo* lineamientos, y en ocasiones habrá razones válidas para no normalizar por completo. El ejemplo clásico de un caso en el cual la normalización completa podría no ser conveniente es la relación de nombres y direcciones

```
NOMDIR ( NOMBRE, CALLE, CIUDAD, ESTADO, CODIGOPOSTAL )
```

en la cual suponemos que, además de las dependencias funcionales implicadas por NOMBRE (la clave primaria), también se cumple la siguiente DF:

```
CODIGOPOSTAL ---> ( CIUDAD, ESTADO )
```

Esta relación no está en 5NF (¿en cuál forma está?), y los lineamientos de normalización antes bosquejados sugerirían la conveniencia de descomponerla en las dos proyecciones

```
NCC ( NOMBRE, CALLE, CODIGOPOSTAL )
CCE ( CODIGOPOSTAL, CIUDAD, ESTADO )
```

(cuyas claves primarias son NOMBRE y CODIGOPOSTAL, respectivamente). Sin embargo, como CALLE, CIUDAD y ESTADO casi siempre se necesitan juntos (pensemos en la impresión de una lista de envíos por correo), y como los códigos postales no se modifican con mucha frecuencia, no es muy probable que una descomposición así valiera la pena.

The Joy* of Denormalization

- To see the “benefits” of denormalization, let us start with a simple example:
- Recall that the order_item entity had the following attributes:

order_item_ID

order_ID

quantity

cost_per_unit

product_ID

* It can become a nightmare!

- An example of an order items document is

{
"order_item_ID ": 834838,
"order_ID": 8827,
"quantity": 3,
"cost_per_unit": 8.50,
"product_ID": 3648
}

- The Products entity has the following attributes:

product_ID

product_description

product_name

product_category

list_price

- An example of a product document is

{
"product_ID": 3648,
"product_description": "1 package laser printer paper. 100% recycled.",
"product_name": "Eco-friendly Printer Paper",
"product_category": "Office supplies",
"list_price": 9.00
}

- If you implemented two collections (of documents) and maintained these separate documents, then you would have to query the order items collection for the order item you were interested in and then query the products document for information about the product.
- You would perform two lookups to get the information you need about one order item.

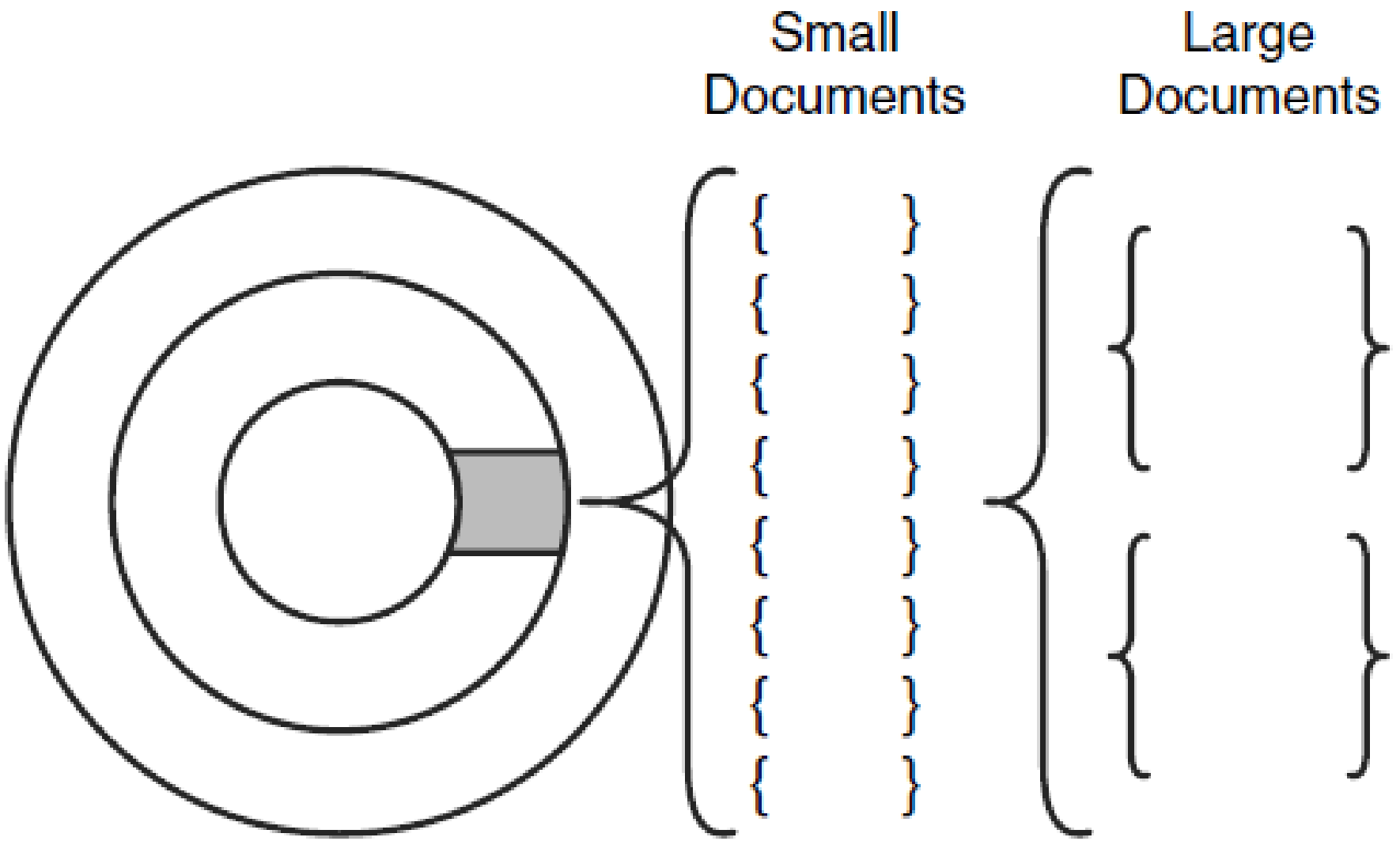
- By denormalizing the design, you could create a collection of documents that would require only one lookup operation.
- A denormalized version of the order item collection would have, for example:

```
{
"order_item_ID": 834838,
"order_ID": 8827,
"quantity": 3,
"cost_per_unit": 8.50,
"product": {"product_description": "1 package laser printer paper. 100% recycled.",
                "product_name": "Eco-friendly Printer Paper",
                "product_category": "Office supplies",
                "list_price": 9.00
              }
}
```

Note Notice that you no longer need to maintain product _ ID fields. Those were used as database references (or foreign keys in relational database parlance) in the Order_Item document.

Avoid Overusing Denormalization

- Denormalization can be used in excess.
- The goal is to keep data that is frequently used together in the document.
- This allows the document database to minimize the number of times it must read from persistent storage.
- Large documents can lead to fewer documents retrieved when a **block of data** is read from persistent storage: *This can increase the total number of data block reads to retrieve a collection or subset of collections.*


Small
Documents
Large
Documents

- To answer the question “how much denormalization is too much?” you should consider the queries your application will issue to the document database.
- Let us assume you will use two types of queries: one to generate invoices for customers and one to generate management reports.
- Also, assume that 95% of the queries will be for invoices and 5% of the queries will be for management reports.

- Invoices should include, among other fields, the following:

- order_ID
- quantity
- cost_per_unit
- product_name

- On the other hand, management reports tend to aggregate information across groups or categories.
- For these reports, queries would include product category information along with aggregate measures, such as total number sold.
- Another usual management report showing the top 25 selling products would likely include a product description.

- Based on these query requirements, you might decide it is better to not store product description, list price, and product category in the Order _ Items collection.
- The next version of the Order _ Items document would then look like this:

```
{
"order_item_ID": 834838,
"order_ID": 8827,
"quantity": 3,
"cost_per_unit": 8.50,
"product_name": "Eco-friendly Printer Paper"
}
```

- We would maintain a Products collection with all the relevant product details; for example:


```
{
"product_description": "1 package laser printer paper. 100% recycled.",
"product_name": "Eco-friendly Printer Paper",
"product_category": "Office supplies",
"list_price": 9.00
}
```

- Product _ name is stored redundantly in both the Order _ Items collection and in the Products collection.
- This model uses slightly more storage but allows application developers to retrieve information for the bulk of their queries in a single lookup operation.

A tradeoff

- Never say never when designing NoSQL models.
- There are best practices, guidelines, and design patterns that will help you build scalable and maintainable applications. None of them should be followed dogmatically
- If your application requirements are such that storing related information in two or more collections is an optimal design choice, then make that choice.

- Normalization is a useful technique for reducing the chances of introducing data anomalies.
- Denormalization is employed to improve query performance.
- When using document databases, data modelers and developers often employ denormalization as readily as relational data modelers employ normalization.

This does not mean that you should forget about avoiding anomalies in a document database or that query performance is prohibitive in a normalized database (this is an implementation problem)

- Remember to use your queries as a guide to help strike the right balance of normalization and denormalization.
- Too much of either can adversely affect performance.
- Too much normalization leads to queries requiring joins.
- Too much denormalization leads to large documents that will likely lead to unnecessary data reads from persistent storage and other adverse effects (redundancies, data anomalies).

- There is another less-obvious consideration to keep in mind when designing documents and collections: the potential for documents to change size. Documents that are likely to change size are known as mutable documents.

Planning for Mutable Documents

- Some documents will change frequently, and others will change infrequently.
- When designing a document database, consider not just how frequently a document will change, but also how the size of the document may change.

Consider the following scenarios:

- Trucks in a company fleet transmit location, fuel consumption, and other operating metrics every three minutes
- The price of every stock: If there is a change since the last check (every minute), the new price information is written to the database.
- A stream of social networking posts is streamed to an application, which summarizes the number of posts and the overall sentiment of the posts

- Over time, the data written to the database increases.
- How should an application designer structure the documents to handle such input streams?
- One option is to create document for each new set of data.
- In the case of the trucks transmitting operational data, this would include a truck ID, time, location data, and so on:

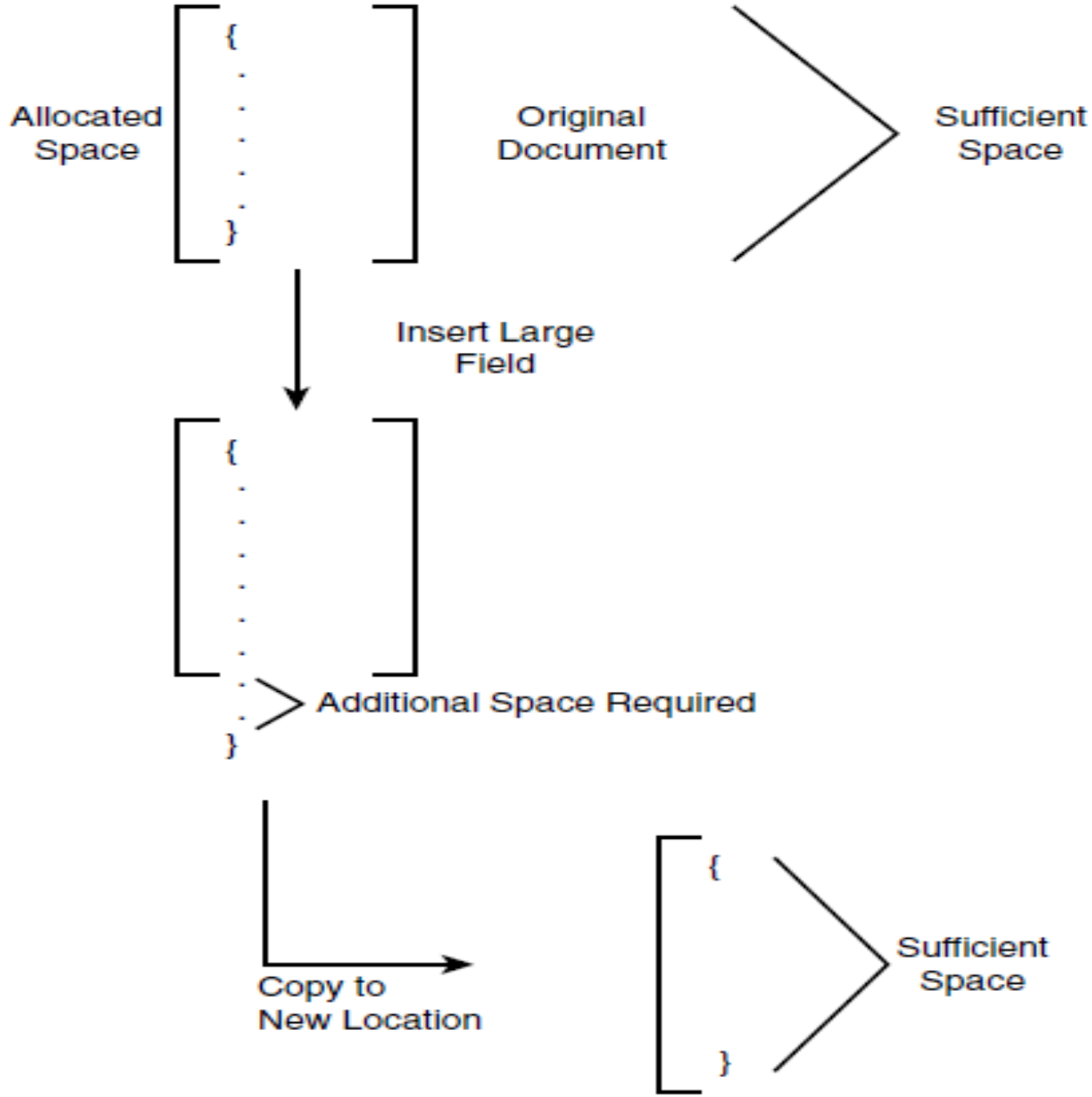
```
{
"truck_ID": "T87V12",
"time": "08:10:00",
"date": "27-May-2015",
"driver_name": "Jane Washington",
"fuel_consumption_rate": "14.8 mpg",
…
}
```

- Each truck would transmit 20 data sets per hour, assuming a 10-hour operations day, 200 data sets per day.
- The truck _ ID, date, and driver_name would be the same for all 200 documents.
- This looks like an obvious candidate for embedding a document with the operational data in a document about the truck used on a particular day.
- This could be done with an array holding the operational data documents:

```
{
"truck_ID": "T87V12",
"date": "27-May-2015",
"driver_name": "Jane Washington",
"operational_data":
[ {"time": "00:01", "fuel_consumption_rate": "14.8 mpg", …},
  {"time": "00:04", "fuel_consumption_rate": "12.2 mpg", …},
  {"time": "00:07", "fuel_consumption_rate": "15.1 mpg", …},
...]
}
```

- The document would start with a single operational record in the array, and at the end of the 10-hour shift, it would have 200 entries in the array.
- From a logical modeling perspective, this is a perfectly fine way to structure the document, assuming this approach fits your query requirements.
- From a physical model perspective; however, there is a potential performance problem.

- When a document is created, the DBMS allocates a certain amount of space for the document.
- If the document grows larger than the size allocated for it, the document may be moved to another location.
- This will require the DBMS to read the existing document and copy it to another location, and free the previously used storage space


Allocated
Space
{
}
Original
Document
Sufficient
Space
Insert Large
Field
{
}
Additional Space Required
Copy to
New Location
{
}
Sufficient
Space

- One way to avoid this problem of moving oversized documents is to allocate sufficient space for the document at the time the document is created.
- In the case of the truck operations document, you could create the document with an array of 200 embedded documents with the time and other fields specified with default values.

- Consider the life cycle of a document and when possible plan for anticipated growth.
- Creating a document with sufficient space for the full life of the document can help to avoid I/O overhead.

Indexes

- When you design a document database, you also want to

  try to identify the right number of indexes.
- You do not want too few, which could lead to poor read performance, and you do not want too many, which could lead to poor write performance.

Read-Heavy Applications

- Some applications have a high percentage of read operations relative to the number of write operations.
- Business intelligence and other analytic applications can fall into this category.
- Read-heavy applications should have indexes on virtually all fields used to help filter results.

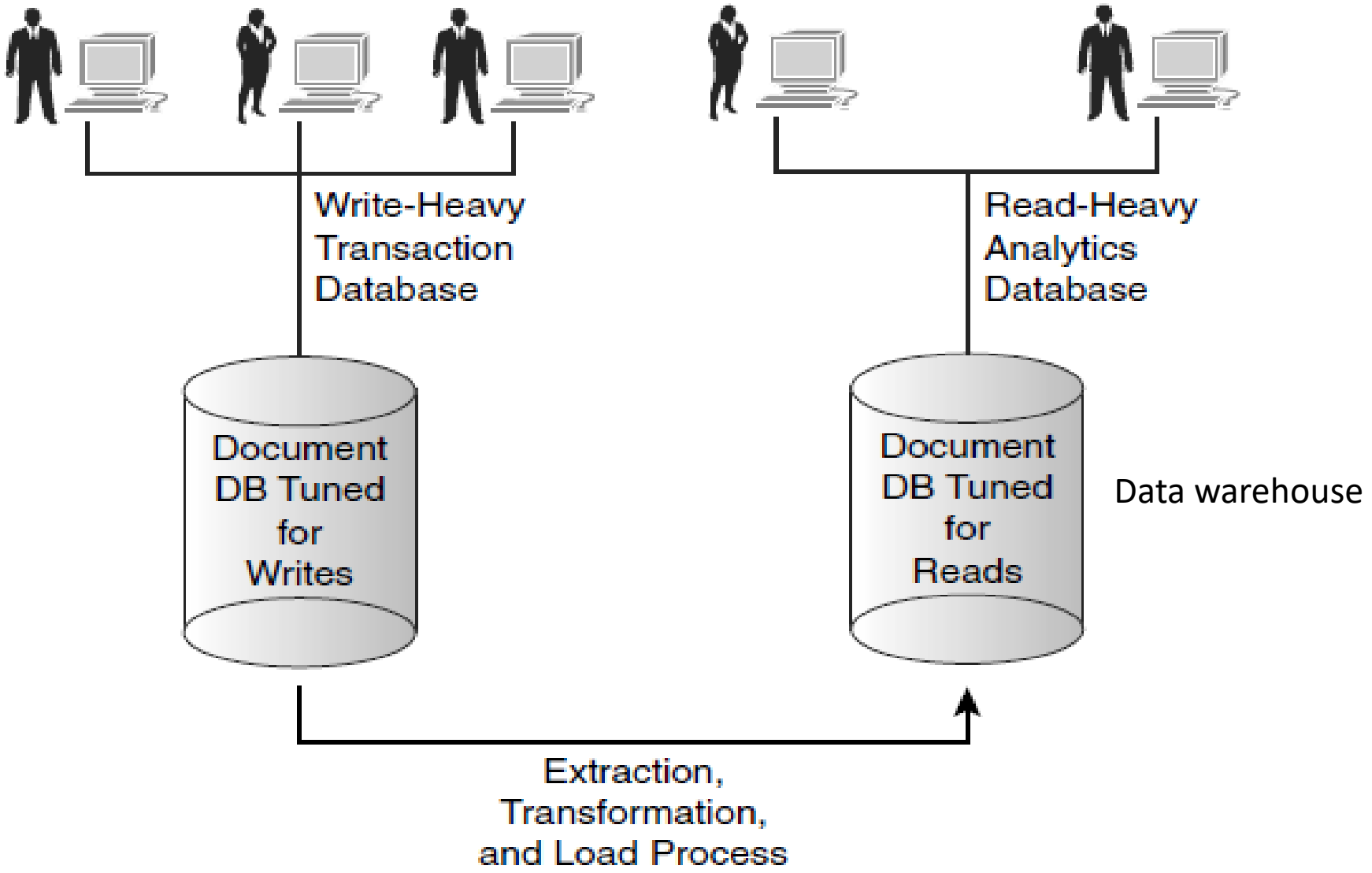
Write-Heavy Applications

- Write-heavy applications are those with relatively high percentages of write operations relative to read operations.
- The document database that receives the truck sensor data described previously would likely be a write-heavy database.
- Because indexes must be updated, their use will consume CPU, persistent storage, and memory resources and increase the time needed to insert or update a document in the database.

Write-Heavy Applications

- Data modelers tend to try to minimize the number of indexes in write-heavy applications.
- Deciding on the number of indexes in a write-heavy application is a matter of balancing competing interests.
- Fewer indexes typically correlate with faster updates but potentially slower reads (affecting queries).

- If users performing read operations can tolerate “some” delay in receiving results, then minimizing indexes should be considered.
- However, if it is important for users to have “fast” queries against a write-heavy database, consider implementing a second database that aggregates the data according to the time-intensive read queries.
- This is the basic model used in business intelligence (where a data mart or a data warehouse is the data core): These two types of databases are usually heavily indexed to improve query response time


Write-Heavy
Transaction
Database
Read-Heavy
Analytics
Database
Document
DB Tuned
for
Writes
Document
DB Tuned
for
Reads
Data warehouse
Extraction,
Transformation,
and Load Process

- Identifying the right set of indexes for your application can take some experimentation.
- Start with the queries you expect to support and implement indexes to reduce the time needed to execute the most important and the most frequently executed.
- If you find the need for both read-heavy and write-heavy applications, consider a two-database solution with one database tuned for each type.

Modeling common relations

As you gather requirements and design a document database, you will likely find the need for one or more of three common relations:

- One-to-many relations
- Many-to-many relations
- Hierarchies

One-to-Many Relations in Document Databases

- This is an example in which the typical model of document database differs from that of a relational database.
- In the case of a one-to-many relation, both entities are modeled using a document embedded within another document.
- For example:

```
{
"customer_id": 76123,
"name": "Acme Data Modeling Services",
"person_or_business": "business",
"address": [{"street": "276 North Amber St", "city": "Vancouver",
                "state": "WA", "zip": 99076} ,
                {"street": "89 Morton St", "city": "Salem",
                "state": "NH", "zip": 01097}
              ]
}
```

One-to-Many Relations in Document Databases

- The basic pattern is that the *one* entity in a one-to-many relation is the primary document, and the *many* entities are represented as an **array of embedded documents** (secondary documents).
- The primary document has fields about the *one* entity, and the embedded documents have fields about the *many* entities.

Many-to-Many Relations in Document Databases

- Many-to-many relations are modeled using two collections—one for each type of entity.
- Each collection maintains a list of identifiers that reference related entities.
- For example, a document with course data would include an array of student IDs, and a student document would include a list of course IDs, as in the following:

Courses:


```
{
{"courseID": "C1667",
"title": "Introduction to Anthropology",
"instructor": "Dr. Margret Austin",
"credits": 3,
"enrolledStudents": ["S1837", "S3737", "S9825", …,"S1847"] },
{"courseID": "C2873",
"title": "Algorithms and Data Structures",
"instructor": "Dr. Susan Johnson",
"credits": 3,
"enrolledStudents": ["S1837", "S3737", "S4321", "S9825“,…, "S1847"] },
…
}
```

Students:

{

{"studentID": "S1837",

"name": "Brian Nelson",

"gradYear": 2018,

"courses": ["C1667", "C2873", "C3876"] },

{"studentID": "S3737",

"name": "Yolanda Deltor",

"gradYear": 2017,

"courses": ["C1667", "C2873"] },

…

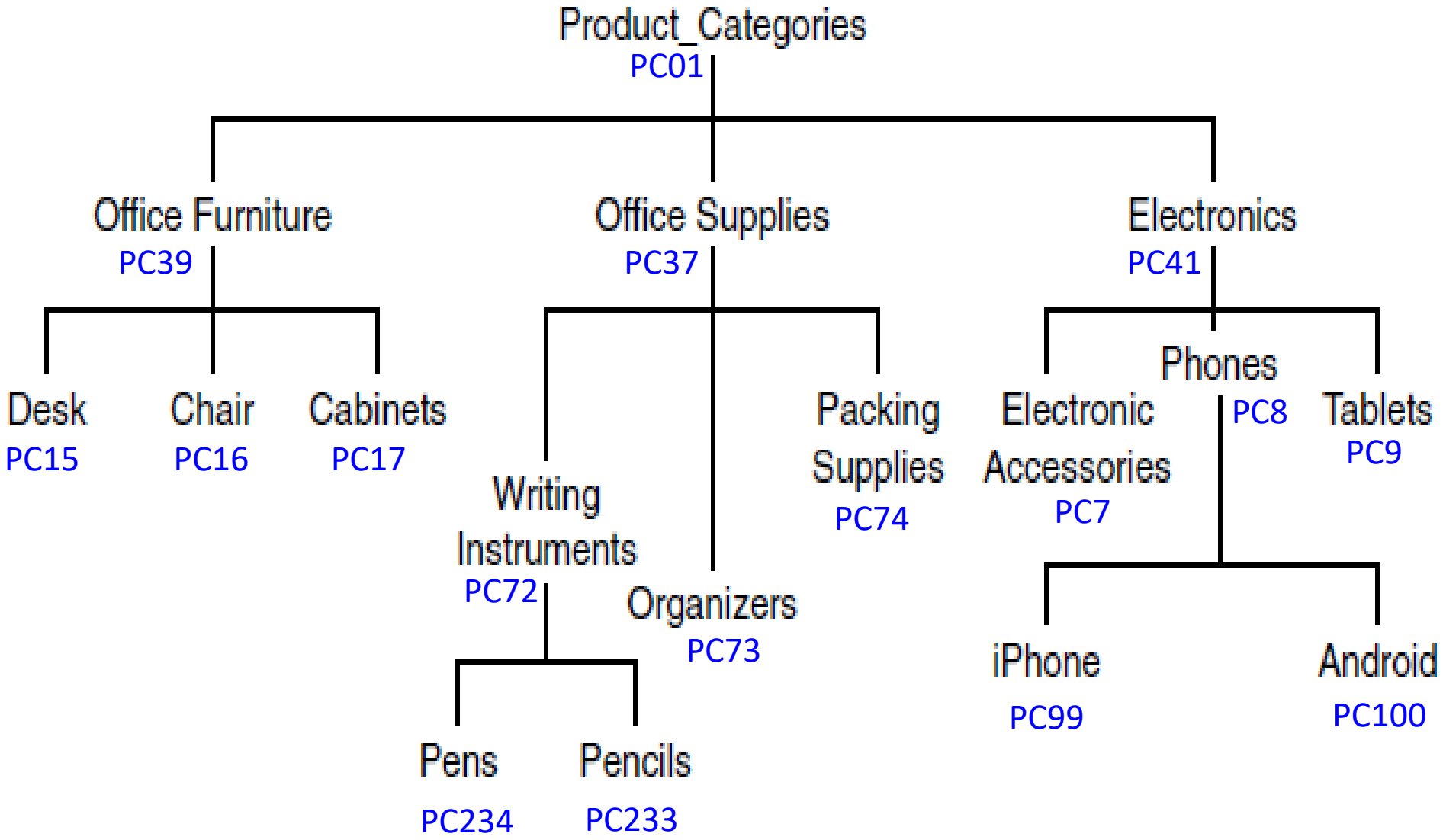
}

Many-to-Many Relations in Document Databases

- The pattern minimizes duplicate data by referencing related documents with identifiers instead of embedded documents.
- Care must be taken when updating many-to-many relationships so that both entities are correctly updated.
- Document databases will not catch referential integrity errors as a relational database will: Document databases will allow you to insert a student document with a courseID that does not correspond to an existing course.

Modeling Hierarchies in Document Databases

- Hierarchies describe instances of entities in some kind of parent-child or part-subpart relation.
- There are a few different ways to model hierarchical relations. Each works well with particular types of queries.
- Consider the following example, a product hierarchy:


Product_Categories
PC01
Office Furniture
PC39
Office Supplies
PC37
Electronics
PC41
Desk
PC15
Chair
PC16
Cabinets
PC17
Writing
Instruments
PC72
Organizers
PC73
Packing
Supplies
PC74
Phones
PC8
Electronic
Accessories
PC7
Tablets
PC9
Pens
PC234
Pencils
PC233
iPhone
PC99
Android
PC100

Modeling Hierarchies in Document Databases

- **Parent or Child References**

A simple technique is to keep a reference to either the parent or the children of an entity.

Using the data depicted in the previous Figure, you could model product categories with references to their parents:

```
{
{"productCategoryID": "PC233", "name": "Pencils",
"parentID": "PC72"},
{"productCategoryID": "PC72", "name": "Writing Instruments",
"parentID": "PC37"},
{"productCategoryID": "PC37", "name": "Office Supplies",
"parentID": "PC01"},
{"productCategoryID": "PC01", "name": "Product Categories" }
…
}
```

- Notice that the root of the hierarchy, 'Product Categories', does not have a parent and so has no parent field in its document.
- This pattern is useful if you frequently have to show a specific instance and then display the more general type of that category.
- A similar pattern works with **child references**:

```
{
{"productCategoryID": "PC01", "name": "Product Categories",
"childrenIDs": ["PC37", "PC39", "PC41"]},
{"productCategoryID": "PC37", "name": "Office Supplies",
"childrenIDs": ["PC72", "PC73", "PC74"]},
{"productCategoryID": "PC72", "name": "Writing Instruments",
"childrenIDs": ["PC233", "PC234"]},
{"productCategoryID": "PC233", "name": "Pencils"}
…
}
```

- The bottom nodes of the hierarchy, such as 'Pencils', do not have children and; therefore, do not have a childrenIDs field.
- This pattern is useful if you routinely need to retrieve the children or subparts of the instance modeled in the document.
- For example, if you had to support a user interface that allowed users to drill down, you could use this pattern to fetch all the children or subparts of the current level of the hierarchy displayed in the interface.

- **Listing All Ancestors**

Instead of just listing the parent in a child document, you could keep a list of all ancestors.

For example, the 'Pencils' category could be structured in a document as:

{"productCategoryID": "PC233", "name": "Pencils",
"ancestors": ["PC72", "PC37", "PC01"]}

- This pattern is useful when you have to know the full path from any point in the hierarchy back to the root.
- An advantage of this pattern is that you can retrieve the full path to the root in a single read operation.
- On the other hand, using a parent or child reference requires multiple reads, one for each additional level of the hierarchy.

- A disadvantage of this approach is that changes to the hierarchy may require many write operations.
- The higher up in the hierarchy the change is, the more documents will have to be updated.
- For example,if a new level was introduced between 'Product Category' and 'Office Supplies', all documents below the new entry would have to be updated.

- The patterns described here are useful in many situations, but you should always evaluate the utility of a pattern with reference to the kinds of queries you will execute and the expected changes that will occur over the lives of the documents.
- Patterns should support the way you will query and maintain documents by making those operations faster or less complicated than other options.