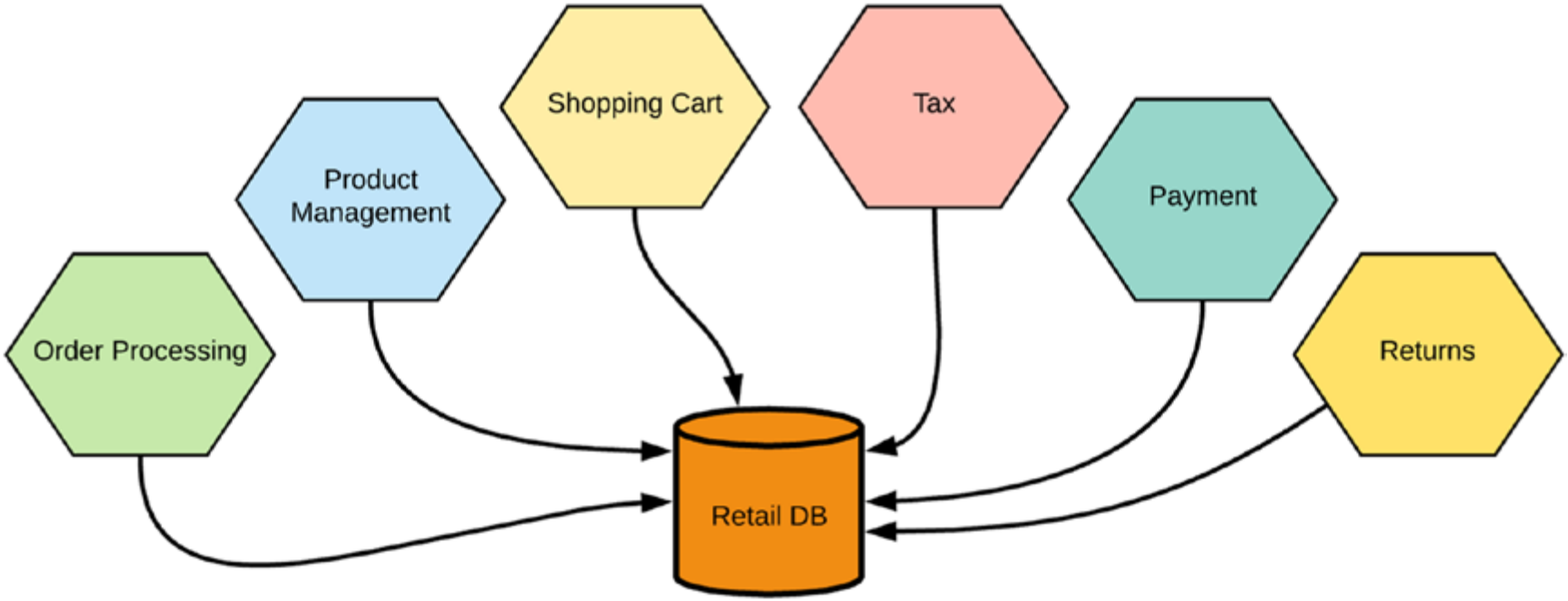
Consistency Models

Shared Databases



Shared Databases

- It is the single point of failure, creates a potential performance bottleneck due to heavy application traffic directed into a single database, and has tight dependencies between applications, as they share same database tables.

- When we share a table between two or more microservices, a change to the schema of that table could affect all dependent microservices.

- So, you cannot build autonomous and independent microservices if you are using a shared persistent layer or database.

Database Per Service

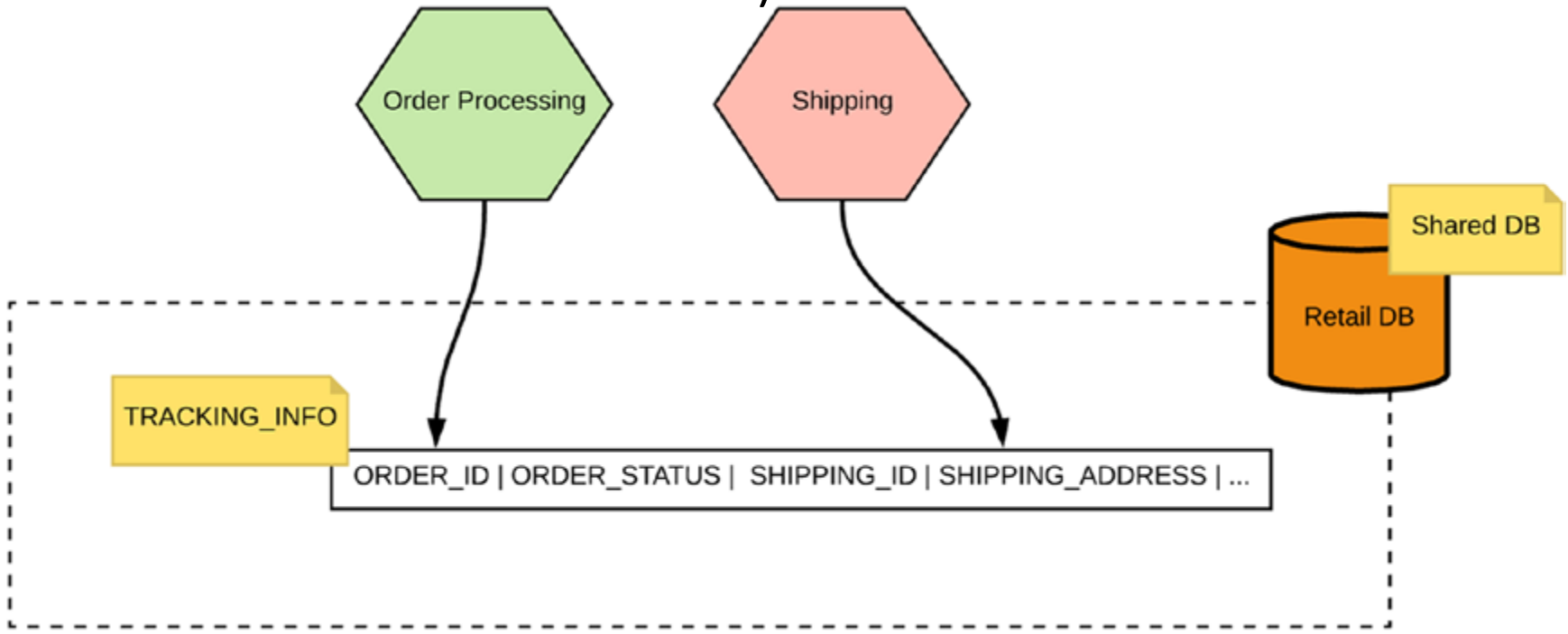
- Having a database per microservice gives us a lot of freedom when it comes to microservices autonomy.
- For instance, microservices owners can modify the database schema as per the business requirements, without worrying about the external consumers of the database.
- There's nobody from the external applications who can access the database directly.
- This also gives the microservices developer the freedom to select the technology (stack) to be used as the persistent layer of microservices.
- Different microservices can use different persistent store technologies, such as RDBMS, NoSQL or other cloud services.

Database Per Service

- In a monolithic database, it is quite easy to do any arbitrary data composition because we share a single monolithic database.
- However, in the microservices context, every piece of data is owned by a single service (single system of record).
- A system of record (or persistent layer) cannot be directly accessed from any other service or system.
- The only way to access the data owned by another microservice is through a service interface or API.
- Other systems, which access the data through the published API, possibly could use a read-only local cache to keep the data locally.

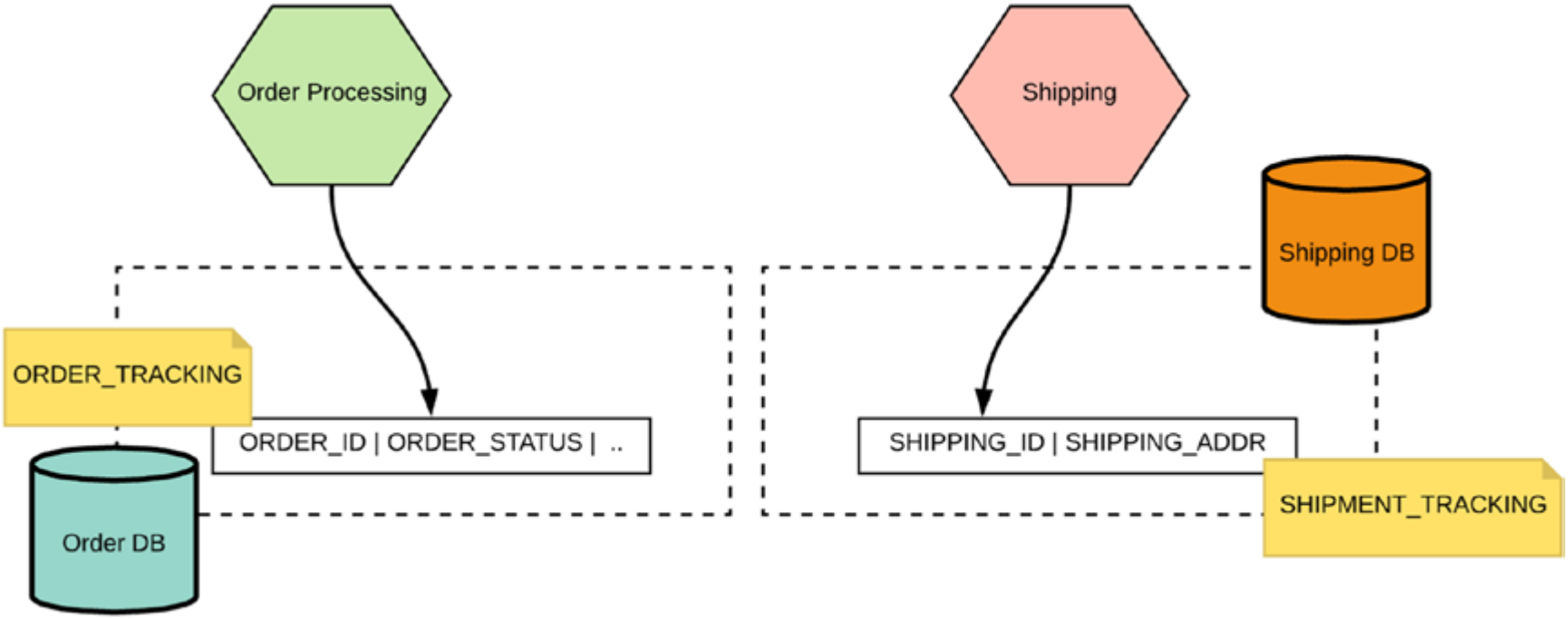
Eliminating Shared Tables

- If we need to change the schema of the TRACKING_INFO table, then that will affect both the Order Processing and Shipping services.
- Also, it is not possible to have service specific data (that service would not like to share) in the shared table.



Eliminating Shared Tables

- There can be shared data duplicated on these two tables and services are responsible for keeping the data in-sync using the published APIs of those services (no direct database access).
- Recall Non-loss Decomposition in DBMS for NFs

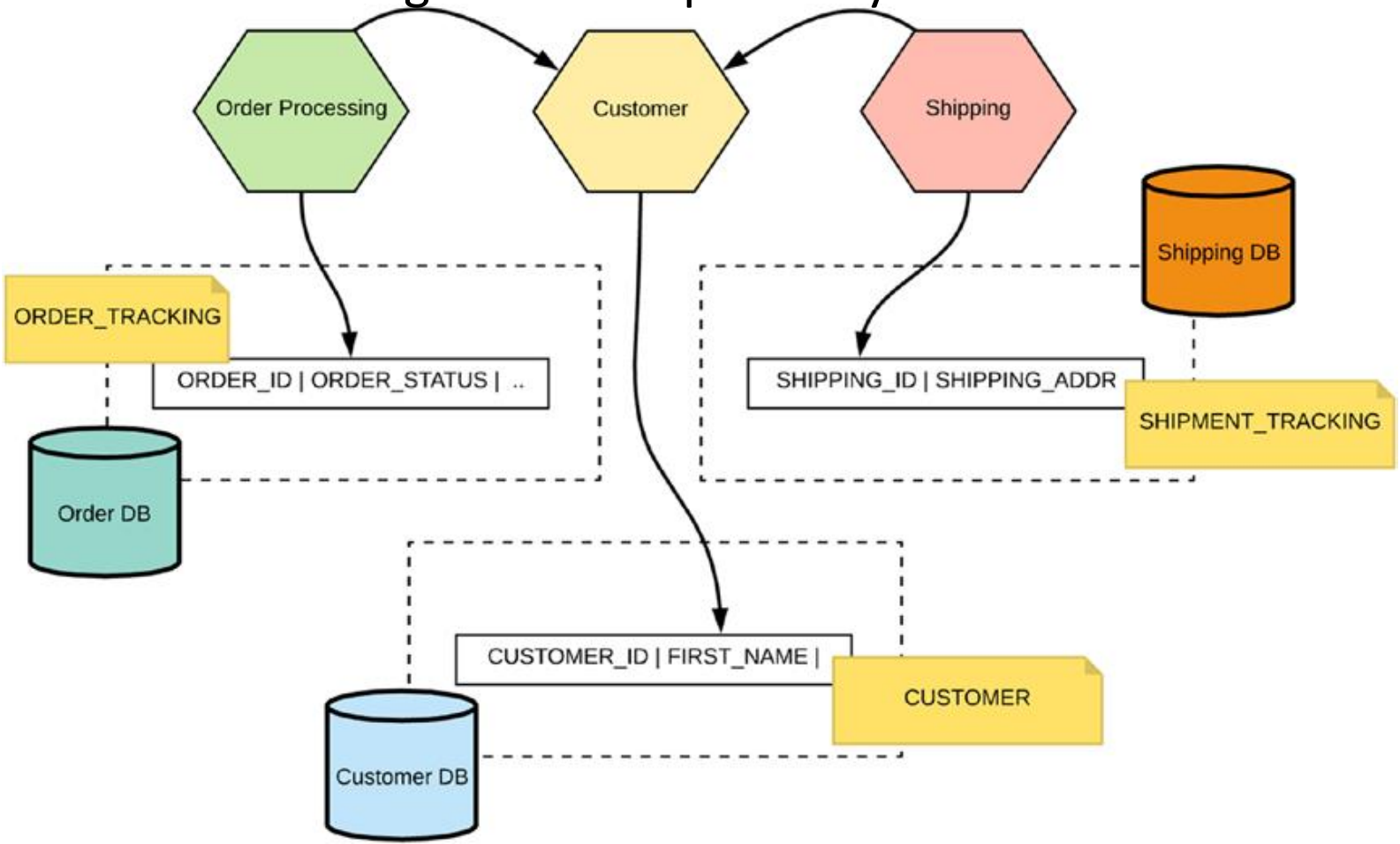


Eliminating Shared Tables

- Another variation of shared tables is when the shared data is represented as a separate business entity.
- In the previous example, the shared data (tracking information) doesn't represent a business entity.
- Example of sharing customer data between the Order Processing and Product Management services.
- In this case, both these services use data from a shared data table (CUSTOMER table) in their business logic.
- We can now identify that customer information is not just a table but also a completely different business entity.
- We can simply treat it as a business capability oriented entity and model that as a microservice.

Eliminating Shared Tables

- Customer service owns the data and the other services can consume it through an API exposed by this Customer service.



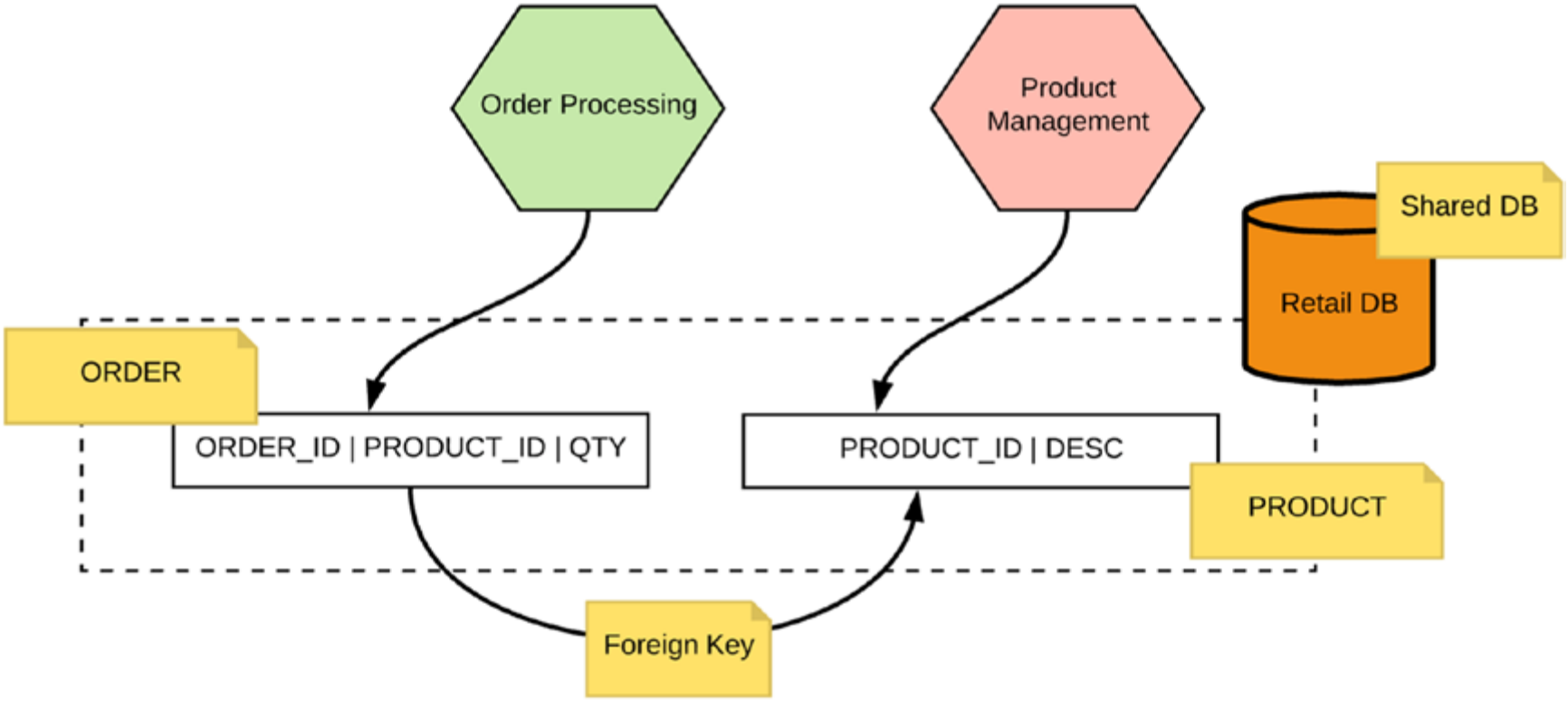
Eliminating Shared Tables

- With this design, dedicated owner of the newly created shared service is necessary that can modify the service interface or schema of that service.
- The key steps involved in eliminating data tables, which share data between multiple microservices:

1. Identify the shared table and identify the business capability of the data stored in that shared table.
2. Move the shared table to a dedicated database and, on top of that database, create a new service (business capability) identified in the previous step.
3. Remove all direct database access from other services and only allow them to access the data via the service's published API.

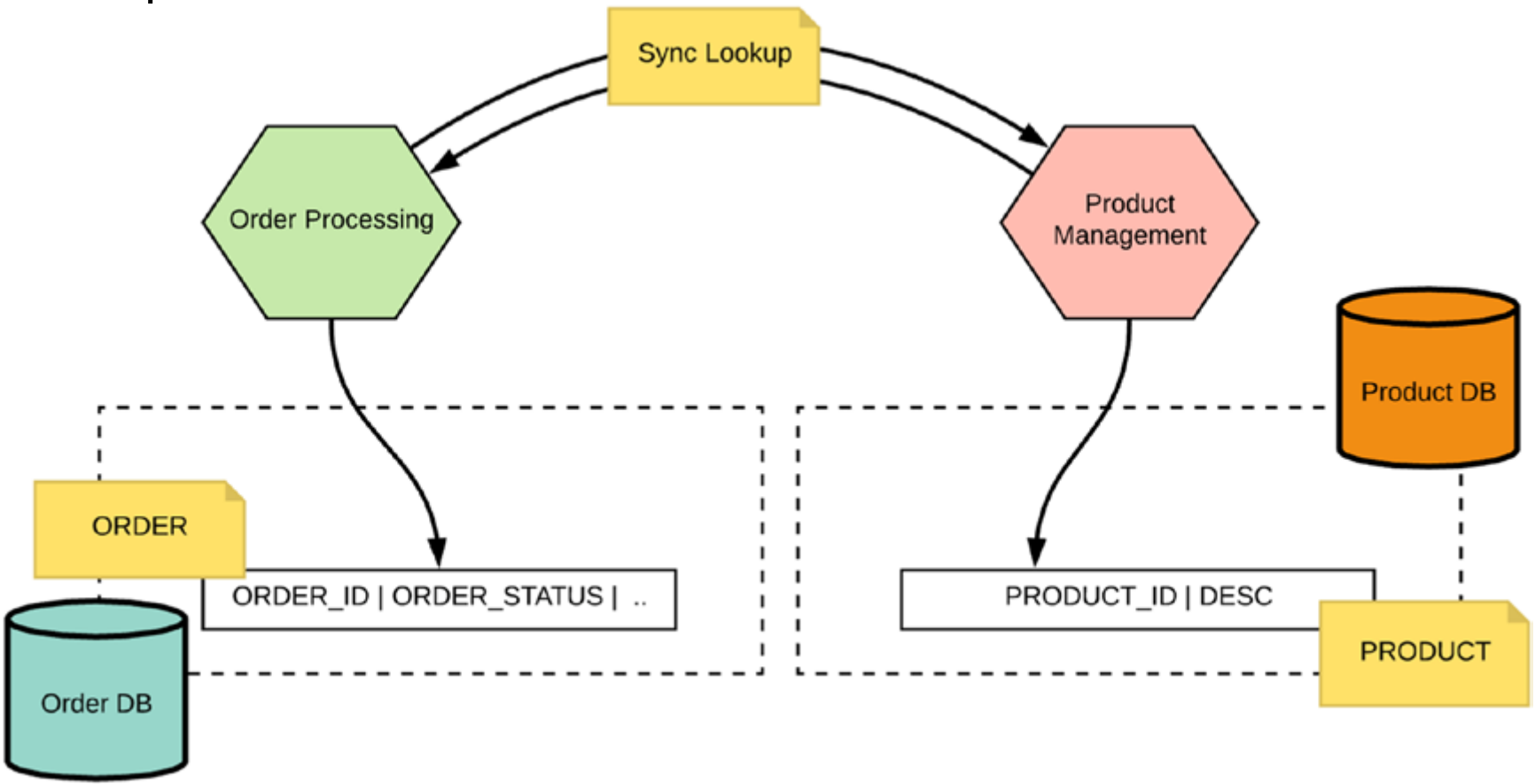
Shared Data

- With monolithic shared databases, using a foreign key and joining data is quite trivial (Referential Integrity Rule)
- For Independent services with DB per service this kind of link for referential integrity is virtually impossible.



Synchronous Lookups

- If one service needs to access the data of the other, it can simply access the published API of that microservice and retrieve the required data.

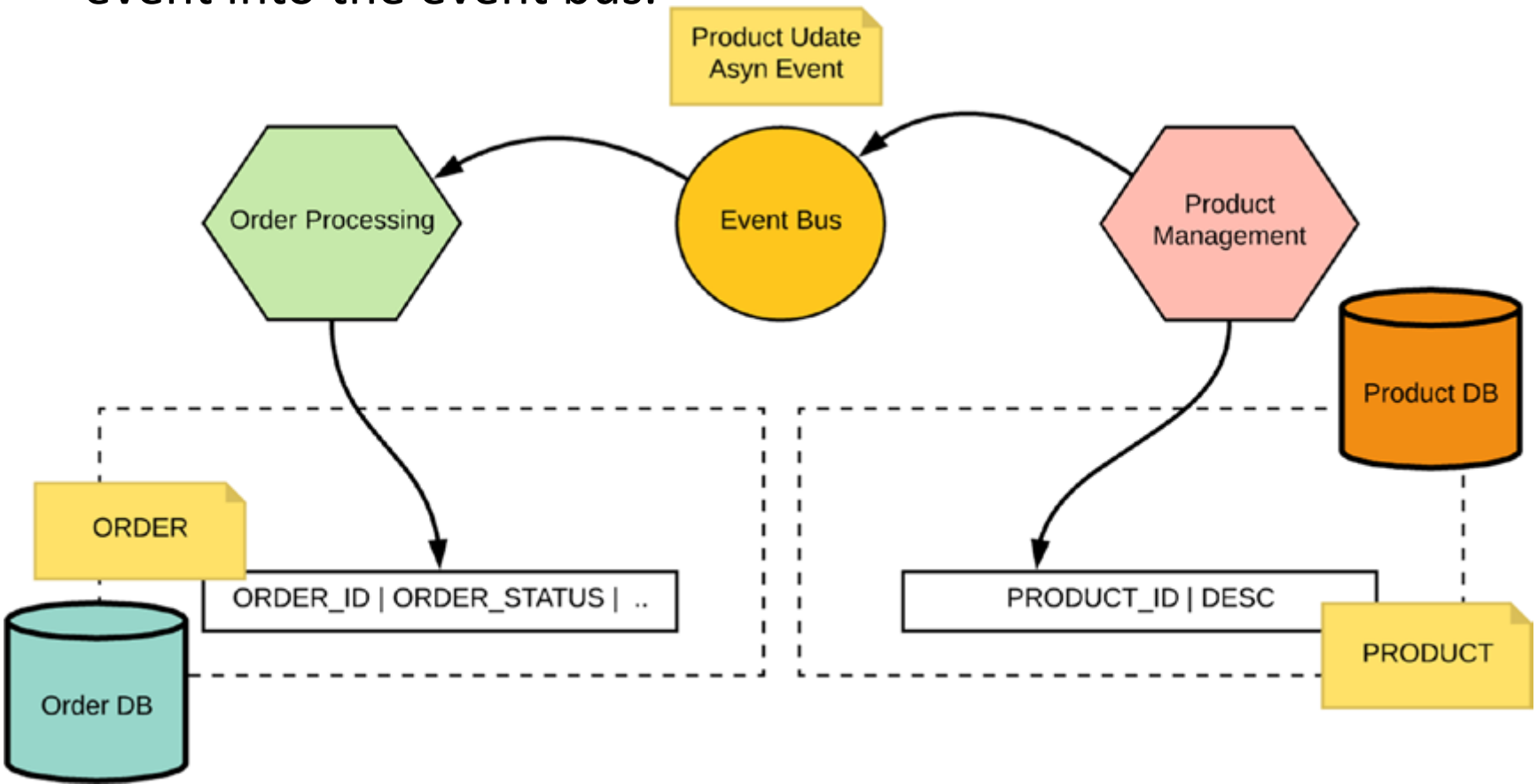


Synchronous Lookups

- This technique is quite trivial to understand and at the implementation level, you need to write extra logic to do an external service call.
- Unlike shared databases, we no longer have the referential integrity of a foreign key constraint.
- This means that the service developers have to take care of the consistency of data that they put into the table.
- For example, when you create an order you need to make sure (possibly by calling the product service) that the products that are referred from that order actually exist in the PRODUCT table

Using Asynchronous Events

- If there is an update to a product, the Product Management service (publisher) updates its product table and publishes an event into the event bus.



Using Asynchronous Events

- The Order Processing service (subscriber) has subscribed to the interested topic of product updates and, therefore, as the Product Management service publishes product update events to that topic, the Order Processing service will receive them.
- Then it can update its local cache of product information and use the cache to implement the business logic related to Product Management service.
- Different subscription techniques to ensure the delivery of the event to the subscriber (such as durable subscriptions).

Data Composition

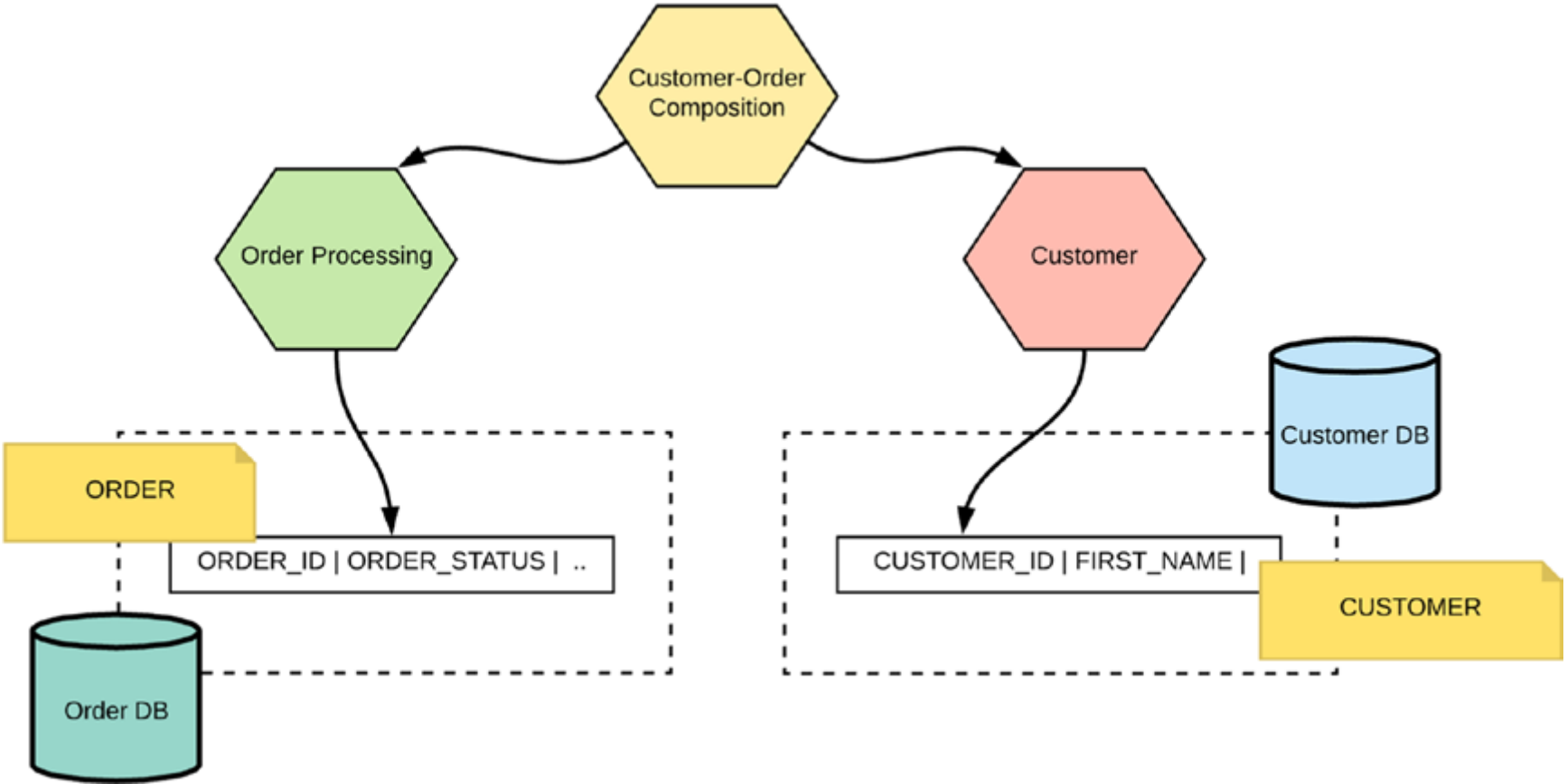
- With monolithic databases (RDBMS in particular), it is trivially easy to build the composition of multiple tables using joins in SQL statements.
- So, you can seamlessly compose different data views out of the exiting entities and use them in your services.
- However, in the microservices context, when you introduce the database per microservice method, building data compositions becomes very complex.
- Built-in constructs such as joins cannot be used directly to compose data, which are dispersed among multiple databases owned by different services.

Data Composition

- When you have to create join of data from multiple microservices, you are only allowed to access the service APIs.
- So, to create composition of data from multiple microservices, you can create a composite service on top of the existing microservices.
- The composite service is responsible for invoking the downstream services and does the runtime composition of the data retrieved via service calls.
- Suppose that we need to **create a composition** of **the orders placed** and have the **details of the customers** who have placed those orders.

Data Composition

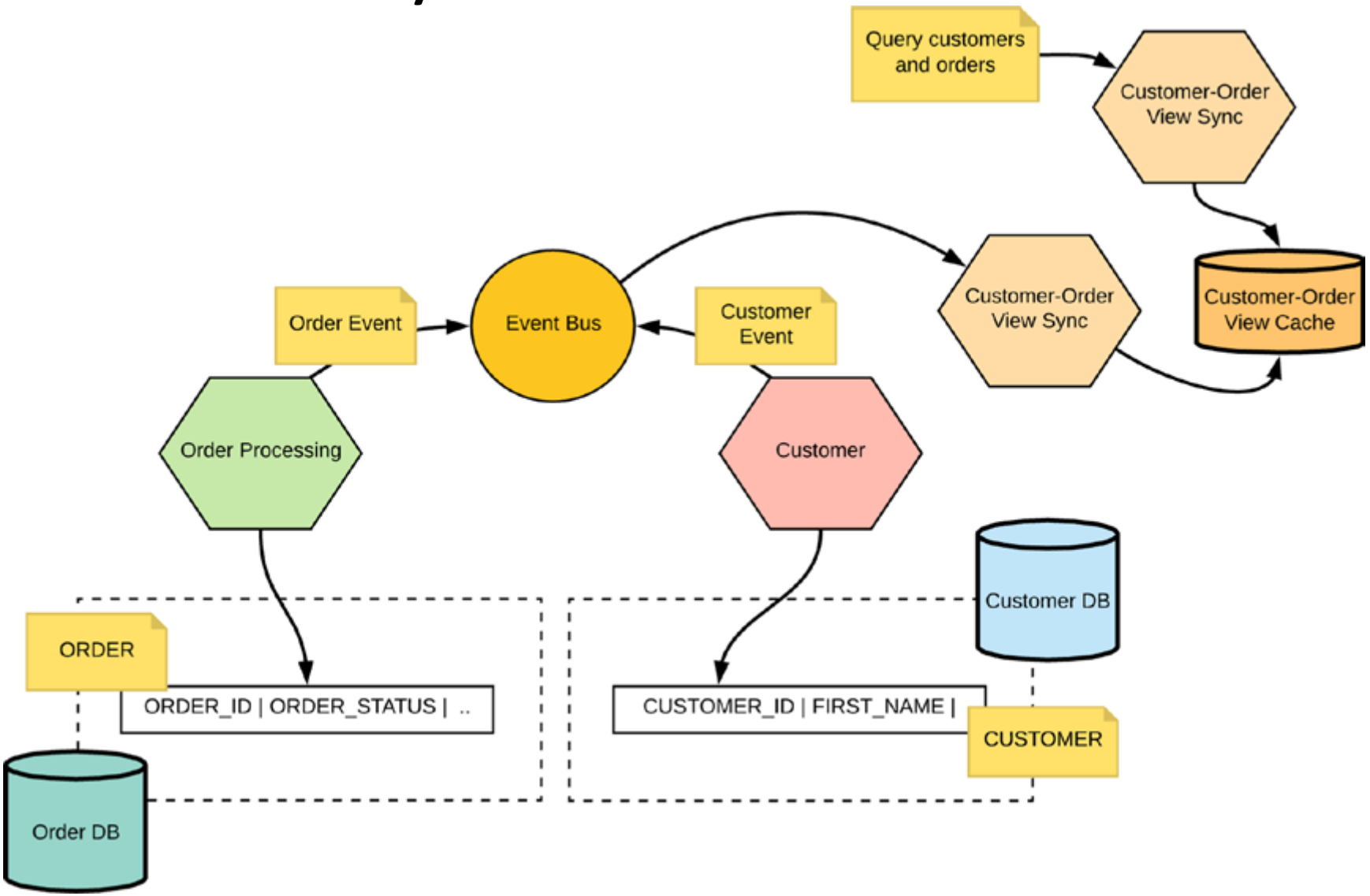
- Need to implement the runtime data composition and communication logic (for example, to invoke RESTful services) to pull required data inside the composition service.



Joins with Materialized View Using Asynchronous Events

- There are certain data composition scenarios where you need to materialize the view with pre-joined data coming from multiple microservices.
- The materialized view will be used for a specific business function like Order Processing and Customer services
- Need to materialize the customer-order join-view.
- A service (Customer-Order-View-Sync) maintains a de-normalized join between the orders and customers, which is done ahead of time rather than on demand in realtime.
- The denormalized data can be kept in a cache or other such storage and it could be consumed by another microservices as a read-only datastore.

Joins with Materialized View Using Asynchronous Events



CONSISTENCY MODELS

The ACID Consistency Model

- Key ACID guarantee is that it provides a safe environment to operate on your data.
- **Atomic**
  - All operations in a transaction succeed or every operation is rolled back.
- **Consistent**
  - On the completion of a transaction, the database is structurally sound, honoring all the integrity constraints.
- **Isolated**
  - Transactions do not contend with one another. Contentious access to data is moderated by the database so that transactions appear to run sequentially (Serializability)
- **Durable**
  - The results of applying a transaction are permanent, even in the presence of failures.

The ACID Consistency Model

- ACID properties mean that once a transaction is complete, its data is consistent and stable on disk, which may involve multiple distinct memory locations.
- Write consistency is a wonderful thing for application developers, but it also requires sophisticated locking which is typically a heavyweight pattern for most use cases.
- When it comes to NoSQL technologies, most graph databases (including Neo4j) use an ACID consistency model to ensure data is safe and consistently stored.

The CAP Theorem

- In a distributed data storage systems with data replication, concurrency control becomes more complex because there can be multiple copies of each data item.

- So if an update is applied to one copy of an item, it must be applied to all other copies in a consistent manner.

- The possibility exists that one copy of an item *X* is updated by a transaction *T*1 whereas another copy is updated by a transaction *T*2, so two inconsistent copies of the same item exist at two different nodes in the distributed system.

- If two other transactions *T*3 and *T*4 want to read *X*, each may read a different copy of item *X*.

The CAP Theorem

- In the field of distributed systems, there are various levels of consistency among replicated data items, from weak consistency to strong consistency.
- Enforcing serializability is considered the strongest form of consistency
- But it has high overhead due to locking needs so it can reduce performance of read and write operations and hence adversely affect system performance.

The CAP Theorem

- The CAP Theorem describes the three architectural properties that are linked together with mutual dependencies as
  - Consistency
  - Availability
  - Partition tolerance
- The theorem states that you can guarantee any two of the three properties.

The CAP Theorem

- *Consistency* means that the nodes will have the same copies of a replicated data item visible for various transactions.

- *Availability* means that each read or write request for a data item will either be processed successfully or will receive a message that the operation cannot be completed.

- *Partition tolerance* means that the system can continue operating if the network connecting the nodes has a fault that results in two or more partitions, where the nodes in each partition can only communicate among each other.

The CAP Theorem

- If you add or increase one of these properties, you are taking away or reducing the other properties.
- In other words, you can have a highly available, consistent system but its partitionability will be low, or you can have a highly available and partitionable system but you would likely need to give up on consistency.

The CAP Theorem

- **Consistency:** All microservice nodes have the same view of the data.
- **Availability:** Every microservice can read or write to any other microservice, all the time.
- **Partition tolerance:** The microservices application as a whole works well despite physical network partitions between the microservices.

The CAP Theorem

- It is important to note here that the use of the word *consistency* in CAP and its use in ACID *do not refer to the same identical concept*.

- In CAP, the term *consistency* refers to the consistency of the values in different copies of the same data item in a replicated distributed system.

- In ACID, the term *consistency* refers to the fact that a transaction will not violate the integrity constraints specified on the database schema.

The CAP Theorem

- It is generally assumed that in many traditional (SQL) applications, guaranteeing consistency through the ACID properties is important.

- On the other hand, in a NOSQL distributed data store, a weaker consistency level is often acceptable, and guaranteeing the other two properties (availability, partition tolerance) is important.

- Hence, weaker consistency levels are often used in NOSQL system instead of guaranteeing serializability.

The CAP Theorem

- When you use a SQL client and issue “Select for Update” or similar commands from the same node where you run your Oracle database, your node acts as a kind of atomic processor in that it either works or it doesn’t
- When you start to separate and spread processing logic from data around different nodes either for scalability or availability, there’s a risk of partitions forming.

The CAP Theorem

- Architecture trade-offs must be made when designing microservices to address the limitations imposed by the CAP Theorem.
- You may design systems without partitions if you have to stop them from network failures.
- In order to do this, you need to put everything related to that transaction on one machine, or in one atomically-failing unit like a single rack.

The BASE Consistency Model

- In NoSQL database world, ACID transactions are less fashionable as some databases have loosened the requirements for immediate consistency, data freshness and accuracy in order to gain other benefits, like **scale and resilience.**

- ACID is based on **pessimistic assumptions** and forces consistency at the end of every operation

- BASE is based on **optimistic assumptions** and accepts that the database consistency will be in a state of flux to a level acceptable to the business transaction in consideration.

The BASE Consistency Model

- **BASE stands for:**
  - **Basically Available**
    - The database appears to work most of the time.
  - **Soft-state**
    - Stores don't have to be write-consistent, nor do different replicas have to be mutually consistent all the time.
  - **Eventual consistency**
    - Stores exhibit consistency at some later point (e.g., lazily at read time).

The BASE Consistency Model

- Define a set of properties for coordination of systems or services as follows:
  - A system is basically available when supporting partial failures, which may be appreciated than total system failure (Graceful Degradation)
  - The state of the system is “soft”, in that it can change over time even in cases where no further updates are made
  - Because some of the past changes are yet to be applied since they are “still on the fly”, originated from other partitions.
  - The system will eventually become consistent if no more new updates are made to the system.

The BASE Consistency Model

- **Example:** Order microservice and Inventory microservice are partitioned, it is possible that even after the last item in stock has been confirmed for Customer A for purchase, Customer B may still add the same item to her shopping cart and confirm check out.
- Customer B will only eventually come to know that this particular item is no longer in stock.
- From a microservices design point of view, these are temporal inconsistencies, which is eventually consistent and hence the temporal unhappiness of Customer B will also eventually be converted to happiness since Customer B will either be informed by the system before payment that the item is no longer available or
- In the worst case where the payment has already been made for a non-available item, a refund will be initiated.

The BASE Consistency Model

- BASE properties << ACID guarantees, but there isn't a one-for-one mapping between the two consistency models
- A BASE data store values availability (since that's important for scale), but it doesn't offer guaranteed consistency of replicated data at write time.
- Overall, the BASE consistency model provides a less strict assurance than ACID: data will be consistent in the future, either at read time or it will always be consistent, but only for certain processed past snapshots
- The BASE consistency model is primarily used by aggregate stores, including column family, key-value and document stores.

ACID vs. BASE Trade-offs

- Developers and data architects should select their data consistency trade-offs on a case-by-case basis – not based just on what's trending or what model was used previously.

- Given BASE's loose consistency, developers need to be more knowledgeable and rigorous about consistent data if they choose a BASE store for their application.

- It's essential to be familiar with the BASE behavior of your chosen aggregate store and work within those constraints.

- On the other hand, planning around BASE limitations can sometimes be a major disadvantage when compared to the simplicity of ACID transactions.

- A fully ACID database is the perfect fit for use cases where data reliability and consistency are essential like OLTP

Two Phase Commit

- A multidatabase transaction, may require access to multiple databases stored on different types of DBMSs
- Each DBMS involved in the multidatabase transaction may have its own recovery technique and transaction manager separate from those of the other DBMSs
- To maintain the atomicity of a multidatabase transaction, it is necessary to have a two-level recovery mechanism.
- A global recovery manager, is required to maintain information needed for recovery, in addition to the local recovery managers and the information they maintain (log, tables).
- Two Phase Lock, Strict 2PL, Rigorous 2PL and Conservative 2 PL are different concepts

Two Phase Commit

- The coordinator usually follows a protocol called the two-phase commit protocol
- Phase 1. When all participating DBs signal the coordinator that the part of the multiDB transaction has concluded, the coordinator sends a message “prepare for commit” to each participant to get ready for committing the transaction.
- Each participating DB receiving that message will force-write all log records and needed information for local recovery to disk and then send OK signal to the coordinator.
- If the force-writing to disk fails or the local transaction cannot commit for some reason, the participating database sends not OK signal to the coordinator.
- After timeout coordinator assumes a not OK response.

Two Phase Commit

- Phase 2: If all participants reply OK, and the coordinator's vote is also OK, the transaction is successful, and the coordinator sends a commit signal to participating DBs
- All the local effects of the transaction and information needed for local recovery is available in the logs of the participating DBs, local recovery from failure is now possible.
- Each participant completes transaction by writing a [commit] entry in the log and permanently updates the DB
- If one or more of the participants or the coordinator have a not OK response, the transaction fails, and the coordinator sends roll back message to UNDO the local effect of the transaction to each participant
- This is done by undoing the local DB operations, using the log

Three Phase Commit

- The biggest drawback of 2PC is that it is a blocking protocol.
- Failure of the coordinator blocks all participating sites, causing them to wait until the coordinator recovers leading to performance degradation, especially if participants are holding locks to shared resources.
- These problems are solved by the three-phase commit (3PC) protocol, which essentially divides the second commit phase into two subphases called prepare-to-commit and commit.

Three Phase Commit

- The prepare-to-commit phase is used to communicate the result of the vote phase to all participants.
- The commit subphase is identical to its two-phase counterpart.
- If all participants vote yes, then the coordinator instructs them to move into the prepare-to-commit state.
- Now, if the coordinator crashes during this subphase, another participant as recovery coordinator can see the transaction through to completion.

Three Phase Commit

- It can verify with other participants if it received a prepare-to-commit message.
- If it did not, then it safely assumes to abort.
- When a participant receives a precommit message, it knows that the rest of the participants have voted to commit.
- If a precommit message has not been received, then the participant will abort and release all locks.
- Thus the state of the protocol can be recovered irrespective of which participant crashes.

Three Phase Commit

- By limiting the time required for a transaction to commit or abort to a maximum time-out period, the protocol ensures that a transaction attempting to commit via 3PC releases locks on time-out.

- The main idea is to limit the wait time for participants who have prepared to commit and are waiting for a global commit or abort from the coordinator.