ECD3 Data Model Design Guide for Distributed Data-Intensive Systems

Susanne Braun

Document Revision 1.0

About this guide

Document Revision: 1.0, February 2021

Inhaltsverzeichnis

ECD3 Data Model Design Guide for Distributed Data-Intensive Systems

1 What is this all about?

This is the ECD3 Data Model Design Guide. It will help you design data models of data-intensive systems in an optimized way and assist you in properly addressing concurrency-control-related design challenges appropriately. You think this should be the job of the database and the infrastructure? Well, this used to be true for quite some time. But data-intensive systems usually call for **high availability and scalability**, which requires data to be distributed and replicated across different computing nodes (usually in the cloud). There is no longer one central system that could mask all the nasty details and pitfalls of concurrently accessing distributed and replicated data. As a consequence, a lot of complexity related to concurrency control and consistency is shifted from the infrastructure layer to the domain layer, and you as a software architect and developer will have to deal with it. Based on a **taxonomy of different types of replicated data**, this guide provides **best practices and design patterns** for optimized data model design. Keeping this classification and best practices in mind will help you design your data model in an optimized way so that the occurrence of write conflicts and potential concurrency anomalies is minimized. In addition to an optimized design, all you need is an operation-based replication framework as outlined in the next section.

2 The ECD3 Replication Framework

What distinguishes the ECD3 replication framework from most other replication frameworks is that it is an **operation-based replication framework with an operation-sending protocol**. It integrates with Domain-Driven Design [1] (DDD). DDD decomposes the application domain into subdomains. Therefore, ECD3 does not consider databases as the units of replication, but **subdomain models**. Figure 1 shows a standard three-layer architecture where domain-driven design is applied [2]. The top layer, the application layer, hosts the coarse-grained business operations that usually span complete user stories. Code-wise, these are implemented using methods of an application service protected by transaction boundaries. Using the dependency inversion principle (short: DIP) [3], transactions are started and ended with the start and the termination of the application service method. Application service methods are the client of the subdomain models hosted in the domain layer. Domain operations of subdomain models are usually implemented in aggregates or domain services. In principle, we distinguish pure read operations (query methods) and updating domain operations (update methods) in the domain model. The domain operations in turn use the infrastructure layer to persist subdomain models. In the infrastructure layer, read and write operations are executed against data items persisted in the database.

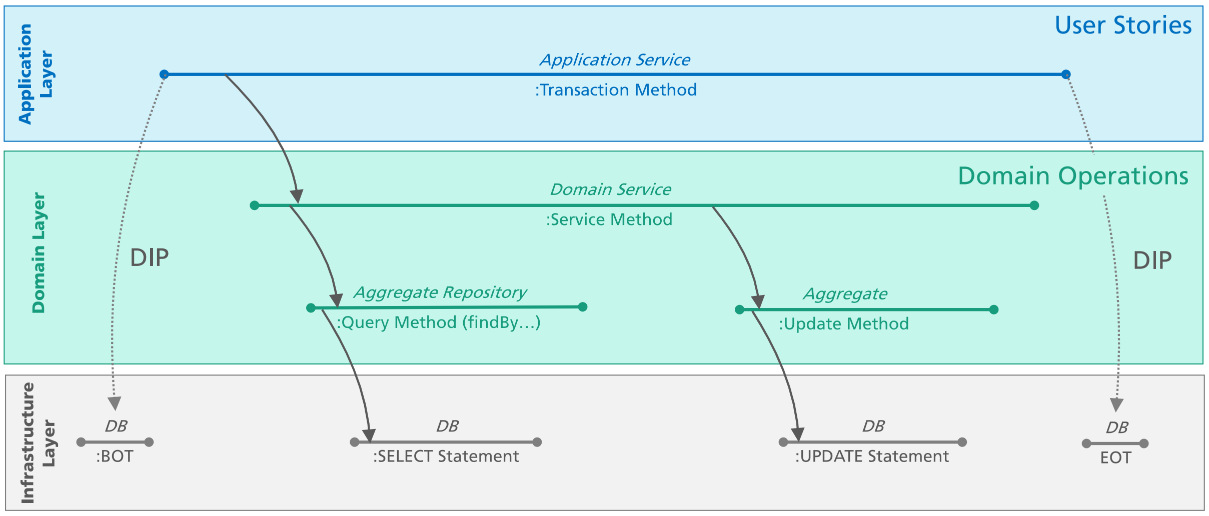


Figure 1 Domain operations in Domain-Driven Design

2.1 Update Propagation & Reconciliation in ECD3

In the following, we will use an example subdomain model from the banking domain to illustrate how updates are propagated and reconciliated with other concurrent updates by the ECD3 framework.

Our sample subdomain model contains an aggregate "`Account`" with the corresponding root entity. The root entity has a property "`balance`" and the domain operations withdraw "`w(amount)`" and deposit "`d(amount)`". Replica$_1$ replicates this subdomain model. The subdomain model holds three account instances: accounts `a`, `b`, and `c`. At some time, Replica$_1$ exchanges updates with other replicas and learns that there have been concurrent domain operation executions at Replica$_2$. A transaction at Replica$_2$ transferred 1000 € from account `c` to account `a`, while at Replica$_1$, a transaction transferred 500 € from account `a` to account `b`. ECD3 replicas maintain an operation log that is organized as a graph as illustrated in Figure 2. Each branch of the graph corresponds to the operation execution history of one replica.

If all concurrent operations are compatible (withdraw and deposit are commutative operations), the concurrent operations can be reconciled by ECD3 with the following reconciliation algorithm:

- Create a new branch for reconciliation
- Copy all operations of the concurrent branches to the reconciliation branch
- Sort operations deterministically on the reconciliation branch so that happens-before-relationships of operations are preserved (version vectors [4] can be used for that)
- Re-execute all operations on the reconciliation branch on a copy of the domain model snapshot that was valid before the replicas diverged
- Persist the new snapshot with the updates executed on the reconciliation branch
- Switch to the reconciliation branch for further execution

After reconciliation, Replica$_1$ will run its local transactions on the reconciliation branch. The old branch will be marked as terminated, as illustrated in Figure 3.

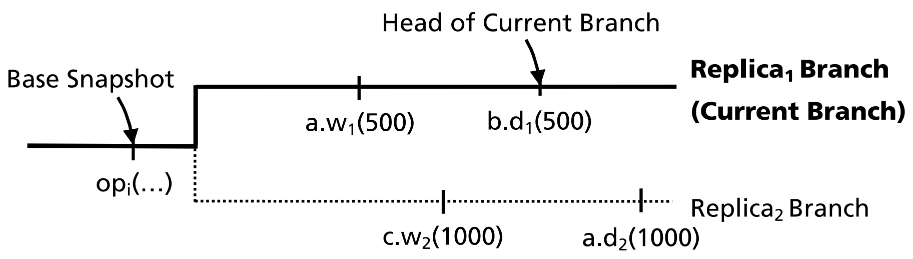


Figure 2 Operation log at Replica$_1$ after update exchange

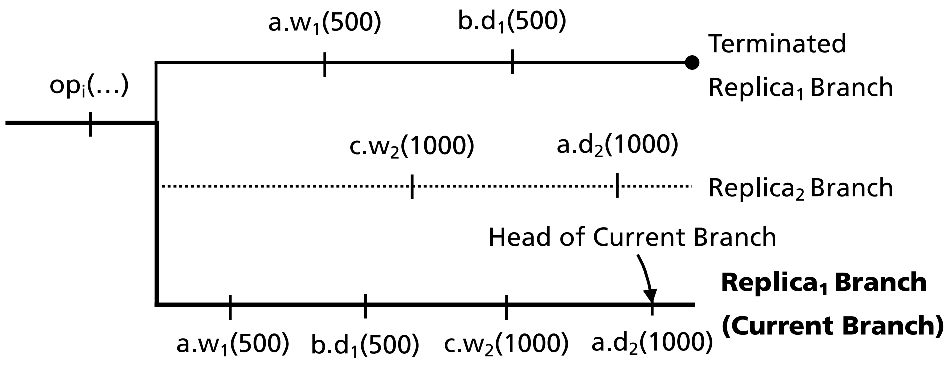


Figure 3 Operation log at Replica$_1$ after reconciliation

By sorting operations according to their happens-before-relationships, ECD3 guarantees causal consistency. The central idea of causal consistency is to preserve causal dependencies between different updates so that updates with causal dependencies are always observed in the proper order by application programs. A causally consistent replicated system ensures that if update u_2 has been executed after u_1, then u_2 is executed after u_1 on all other replicas as well [5]. For example, a comment on a task description in a task management system should only be observed if the task description can be observed as well. Note that we provide a deterministic algorithm for sorting operations on the reconciliation branch so that eventually all operations will be re-executed in the same execution order on all replicas. Thus, strictly speaking, the ECD3 framework guarantees sequential consistency, which is even stronger than causal consistency [6].

3 The ECD3 Aggregate Taxonomy

The ECD3 taxonomy of replicated data considers aggregates (aggregate definition according to Domain-Driven Design[1]) as unit of classification. Aggregates are object graphs composed of entities and value objects. Each aggregate has a root entity. Aggregates are always loaded into the domain layer as one unit and persisted as one unit into the infrastructure layer (e.g., a relational DB or a NoSQL DB) as illustrated in Figure 1. As Domain-Driven Design suggests that each transaction should update at most one aggregate instance, aggregates therefore mark consistency borders. There might be cross-references between different aggregates. An aggregate uses the ID of the root entity to cross-reference another aggregate (see Figure 4).

[1] Vaughn Vernon, Domain-Driven Design Distilled, Addison-Wesley Professional, 2016, https://www.amazon.de/dp/0134434420/ref=cm_sw_em_r_mt_dp_gCXJFbEEEZDQ1

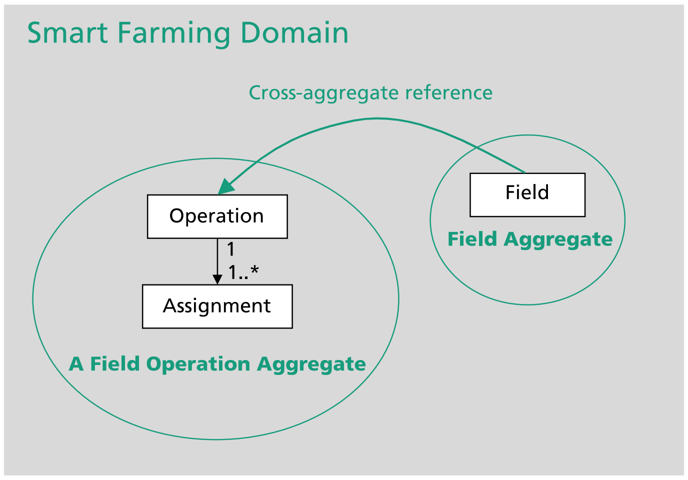


Figure 4 Cross-aggregate reference in a Smart Farming subdomain model

In the ECD3 taxonomy, **mutability** or rather **immutability** of aggregates is an important criterion. We use the immutability definition of Eric Evans, who defines it as the "property of never changing observable state after creation" [1]. In contrast, an aggregate is **mutable** if it can have observable state changes after creation. We define observable state changes of aggregates according to the following definition:

Def.: Observable State Change of an Aggregate

The **observable state of an aggregate changes** if at least one attribute of one of the aggregate's domain objects (entity or value object) is changed. Domain-Driven Design considers the following domain object attribute types:

- primitive values
- references to value objects
- references to entities
- references to other aggregates (by ID of the root entity of the referenced aggregate)

Thus, depending on the attribute type, an attribute change is characterized by the following criteria:

- primitive values attribute type: values have changed / are different
- references to value objects attribute type: at least one different value of at least one referenced value object, or the number of references is different
- references to entities attribute type: identity of at least one referenced entity is different, or the number of references is different
- references to aggregates attribute type: root entity ID of at least on referenced aggregate is different, or the number of references is different

Another important classification criterion is whether aggregates are **updated in place**. An aggregate is updated in place if it is loaded from the infrastructure layer into the domain layer and an (updating) domain operation performs an attribute update directly on the aggregate loaded into main memory. Correspondingly, an attribute update is defined as given in the following:

Def.: Attribute Update

A domain operation performs an **attribute update** if, as a result of the domain operation execution, the observable state of an attribute has changed. Depending on the attribute type, the observable state of a domain object attribute is changed if:

- primitive values attribute type: values have changed / are different
- references to value objects attribute type: value objects have at least one different value, or the number of references is different
- references to entities attribute type: referenced entities have a different identity, or the number of references is different
- references to aggregates attribute type: root entity IDs are different, or the number of references is different

The connection between updating domain operations and attribute updates is illustrated in Figure 5: Updating domain operations (1) reside in the domain layer, (2) are invoked by application service methods in the context of transactions, and (3) perform a number of updates on different object attributes of various aggregates loaded into the domain layer.

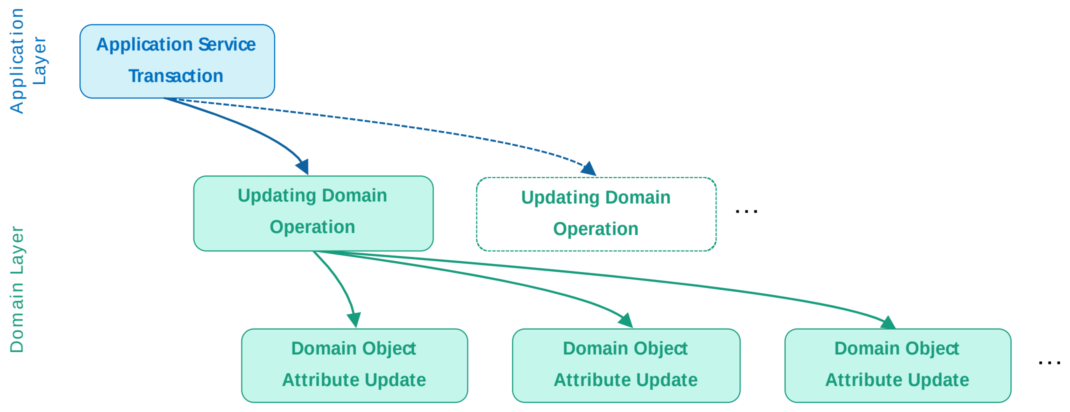

Application Layer
Application Service Transaction
Domain Layer
Updating Domain Operation
Updating Domain Operation
...
Domain Object Attribute Update
Domain Object Attribute Update
Domain Object Attribute Update
...

Figure 5 Updating domain operations in ECD3

In the following, we will introduce the ECD3 aggregate taxonomy, dedicating one section to each aggregate class. Each section provides:

- an informal characterization of aggregates belonging to this class
- unique classification criteria
- illustrative examples
- guiding questions that will help you classify the aggregates of your subdomain models.

The ECD3 data classification distinguishes between trivial aggregates and non-trivial aggregates, as illustrated in Figure 6. We will start with the description of trivial aggregates.

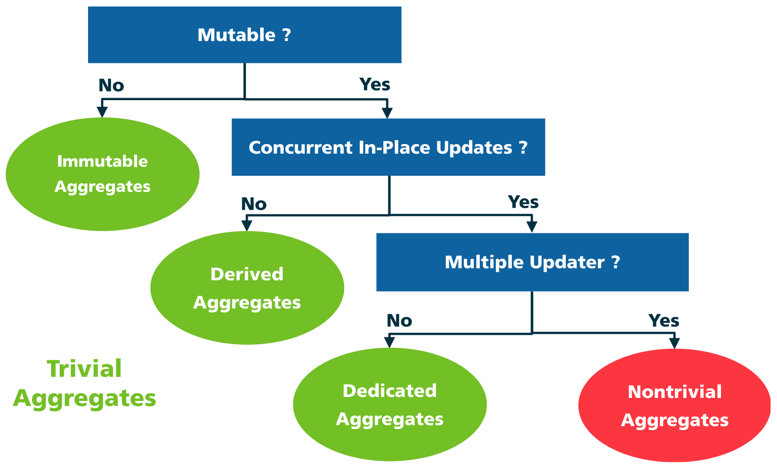


Figure 6 ECD3 distinguishes trivial and non-trivial aggregate classes

3.1 Trivial Aggregates

Trivial aggregates are easier to handle in replicated settings with eventual consistency, so you should try to design your aggregates as trivial aggregates whenever feasible and possible.

3.1.1 The Immutable Aggregates Data Class

Replicated aggregates can be designed as immutable if it can be ensured that after initialization, the aggregate will definitely not require any further changes to its observable state.

If, for whatever reason, the system maintains multiple versions of this aggregate, there must be no differences in the observable state of these versions (meaning that "the versions are the same"). An immutable aggregate can only be "changed" by being completely replaced with another aggregate **of different identity**.

Often, immutable aggregates correspond to what Pat Helland describes as "observed facts" - facts that are observed within a system [7] and never changed. These usually require being recorded for "a very long time" and are therefore not deleted; they may be archived after some time [7]. A good example of observed facts are domain events that are created, recorded, and processed in the system, but are never updated or deleted, such as an order confirmation in an online store. Instead of being updated or deleted, domain events are rather compensated by compensating domain events, like an order cancellation in the example of the online store.

Classification Criteria

Immutability (no observable state changes required after initialization).

Examples

Domain Events

- Order confirmations in e-commerce systems
- Incoming or outgoing goods receipts in warehouse management systems

Immutable Business Data

- Bookings in accounting systems
- The design of a survey that has gone live

Time-Series Data

- Machine sensor values in Industrie 4.0 systems or predictive maintenance systems
- Stock prices in finance systems

Raw Data / Primary Data

- Tracking data used as input for recommendation systems
- Audio recordings used as input data for the training of digital assistants

Guiding Questions

If you can answer some of these questions with "yes", this is an indicator that the aggregate is an observed aggregate:

- Is there a domain invariant demanding immutability of the aggregate?
- Does it not make sense from a business perspective to change the aggregate once it has been initialized and persisted?
- Does the aggregate hold data that needs to be captured and recorded immutably in persistent storage for "a very long time"?

Antipattern: Pointless Use of Append-Only-Storage Principles

Note that **changes to the observable state** of an aggregate do not necessarily need to be implemented in a straightforward way with in-place updates. These could also be realized by means similar to '**multi-versioning'** applied by append-only-storage systems in order to achieve **immutability of data records**. Instead of loading the aggregate into the domain layer, executing attribute updates directly on the loaded aggregate (in-place update), and saving the aggregate back to persistent storage, the following can be done: A new version of the aggregate is created and initialized with the current state of the aggregate; the attribute updates are executed on the new version; the new version is persisted and returned to any subsequent read access. Applying this technique would actually result in **immutable aggregate versions** but not immutable aggregates. The aggregate would still have observable state changes and would thus be mutable. From the standpoint of replication, this is merely a special implementation variant for actually executing in-place updates on an aggregate. It does not help to prevent concurrency anomalies, as different versions of the same aggregate created concurrently on different replicas are still in conflict. In particular, this technique cannot prevent concurrent aggregate versions from shadowing each other's changes or even stopping the occurrence of lost updates.

How to avoid this?

- Do not mix up true immutability of aggregates indicated from the business semantics with technical concepts developed for optimizing write access times for special types of data like time-series data in append-only stores.

3.1.2 The Derived Aggregates Data Class

Derived aggregates are **mutable** as state changes can be observed by the application and the user. At the same time, derived aggregates are immutable from a technical perspective, as in the domain layer, domain operations do **not** directly perform attribute updates on the aggregate (**no in-place updates**). Rather, the current state of derived aggregates is always calculated on demand based on other input data. Only for performance reasons might it be pre-calculated periodically. This class is also based on Helland's append-only-computing pattern and corresponds to "derived facts" in his parlance.

Classification Criteria

Mutability (observable state changes after initialization), but **no in-place updates** (domain operations do not perform attribute updates directly on the aggregate). Rather, the current aggregate state is always calculated (on-demand or periodically) based on other input data.

Examples

Calculations and Aggregations

- Data warehouse reports in data warehousing systems
- Metrics and KPIs in dashboards
- Number of in-stock items in e-commerce systems (compare to Figure 7)

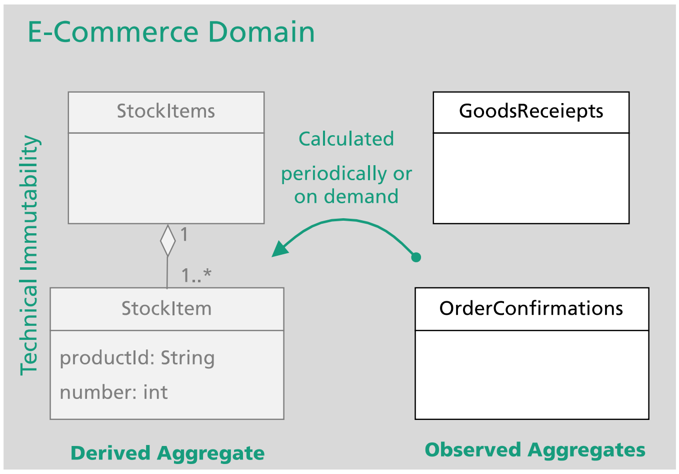


Figure 7 Derived aggregate from the E-Commerce domain

Machine Generated Data

- Recommendations in online-shops, streaming service providers, or booking systems
- Machine Learning models in AI systems
- Predictions in forecasting systems like demand predictions, cost predictions, weather predictions, …

Personalized / Contextual Real-Time Data

- Timelines in social media systems
- Newsfeeds or activity streams in collaboration systems

Guiding Questions

If you can answer some of these questions with "yes", this is an indicator that the aggregate is an observed aggregate:

- Can the current state of the aggregate at any time be derived by evaluating other available data?
- If not, could the current state of the aggregate in principle be derived based on other data that could be collected and recorded by the system?

3.1.3 The Dedicated Aggregates Data Class

Dedicated aggregates are **mutable** and can receive **concurrent in-place updates**, but their particularity is that these concurrent updates are always issued by the same updater **to which the aggregate is dedicated**. An example is the settings data of a particular user account in a cloud service. The user might use different devices to access the cloud service and make changes to their settings. If one of these devices has connectivity issues or synchronizes updates infrequently, concurrent updates to the settings data might occur and can result in lost updates. This can be annoying to the user, but at least it is usually transparent to them what has happened.

Classification Criteria

Mutability (observable state changes after initialization) and **concurrent in-place updates** (domain operations perform attribute updates directly on the aggregate), but there is only a **single updater** (user or system) to which the aggregate is dedicated.

Examples

User-Generated Data

- Reviews in online shops, app stores, or travel portals
- Posts, comments, reactions in social media systems
- Chat messages in messaging applications

Dedicated Master Data

- Personal account settings in apps or cloud services
- Personal user profiles in online platforms and social media

Configuration Data

- Configuration data of a particular system in a distributed system

Guiding Questions

If you can answer some of these questions with “yes”, this is an indicator that the aggregate is an observed aggregate:

- Is there only one updater (user, account, or system) that has the authority to perform updates on the aggregate?
- Do the contents of the aggregate represent “crowd data” (data generated by a large number of users that is collected and utilized by a platform to generate additional values and services)?

3.2 Non-Trivial Aggregates

Non-trivial aggregates are mutable and are concurrently updated by multiple updaters, which makes it so challenging to handle them under eventual consistency. We classify non-trivial aggregates based on their update frequency as well as their probability of simultaneous updates, as well as their potential for write conflicts and concurrency anomalies. In our experience, non-trivial aggregates typically fall into three different classes in practice, which we will introduce in the following.

3.2.1 The Reference Aggregates Data Class

Reference aggregates are **long-lived, relatively stable aggregates** that are rarely updated and, if so, usually only by a small number of different updaters. Even though reference aggregates are seldom updated, they frequently contain business-critical data that is often referenced by a lot of other aggregates. Reference data is often core data of the domain and therefore also referenced by a lot of systems of the overall business domain, meaning that it is usually replicated to a lot of different systems.

Classification Criteria

The aggregate is non-trivial and therefore receives concurrent in-place updates from multiple updaters.

Update frequency at peak times or inconsistency windows	Probability of simultaneous updates at peak times or inconsistency windows	Probability of write conflicts and concurrency anomalies
low	low	low

Examples

Master Data

- Customer data, e.g., in customer relationship management systems
- Resources, products, and assets, e.g., in enterprise resource management systems
- Product descriptions in e-commerce systems

Values

- Valid currencies, product types, and gender

Meta Data

- Tags and descriptive data for the interpretation of raw data

Guiding Questions

If you can answer some of these questions with “yes”, this is an indicator that the aggregate is an observed aggregate:

- Does the aggregate represent classical master data that is referenced throughout the whole system?
- Is the aggregate long-lived and rarely updated?
- Is it unlikely that updates will be performed by different updaters at the same time?
- Is the aggregate a single value object or a composed value?

- Does the aggregate represent meta data?

3.2.2 The Activity Aggregates Data Class

Activity aggregates typically encapsulate the **state of activities** with **multiple actors**. Actors perform actions or sub-activities that trigger in-place updates on the state of the activity. Activity aggregates are often **rather short-lived** in comparison to reference aggregates, but have a lot of state changes during their lifetime. While the activity is running, this can, during peak times, result in simultaneous in-place updates on the state of the activity.

Classification Criteria

The aggregate is non-trivial and therefore receives concurrent in-place updates from multiple updaters.

Update frequency at peak times or inconsistency windows	Probability of simultaneous updates at peak times or inconsistency windows	Probability of write conflicts and concurrency anomalies
medium - high	**medium - high**	**medium - high**

Examples

Business Processes and Workflows

- State of an order in an online store
- State of a package delivery
- Booking status of limited resources like airplane seats, meeting rooms, hotel rooms
- State of an online auction

Coordination Data

- An agricultural operation performed by multiple machines and operators on the field
- State of an online Kanban board used by teams for self-organization
- Tasks in a task management system

Guiding Questions

If you can answer some of these questions with “yes”, this is an indicator that the aggregate is an observed aggregate:

- Does the aggregate represent the state of a business process or workflow that has multiple actors?
- Does the aggregate represent the availability status of a limited resource with different actors racing for that resource?
- Does the aggregate contain data that multiple actors use to coordinate some joint activity (often in real time)?
- Does the aggregate contain business operations that are not trivial (a trivial update operation would be a setter method)?
- Are there invariants associated with the activity?

3.2.3 The Collaboration Result Aggregates Data Class

Collaboration result aggregates encapsulate data produced through the collaboration of multiple updaters and usually contain the results of knowledge-generating work. Please note the interesting case that updaters can also be bots that generate, e.g., the first draft of a meeting protocol. Even though collaboration result aggregates can be either long-lived or short-lived, there are normally peak times with a very high number of simultaneous in-place updates executed on the aggregate; for example, at the end of an agile sprint when the team finishes some documentation artifact together.

Classification Criteria

The aggregate is non-trivial and therefore receives concurrent in-place updates from multiple updaters.

Update frequency at peak times or inconsistency windows	**Probability of simultaneous updates at peak times or inconsistency windows**	**Probability of write conflicts and concurrency anomalies**
medium – very high	**high - very high**	**high - very high**

Examples

Work Results

- A CAD model in a CAD system
- A software component model in a software architecture modeling system
- A machine learning model in a data science notebook
- A crop rotation plan in a smart farming system
- A spreadsheet calculation
- A whiteboard diagram

Knowledge

- Manuals and tutorials in wiki systems
- Scientific papers in collaborative text editing systems
- User story descriptions in backlog systems

Guiding Questions

If you can answer some of these questions with "yes", this is an indicator that the aggregate is an observed aggregate:

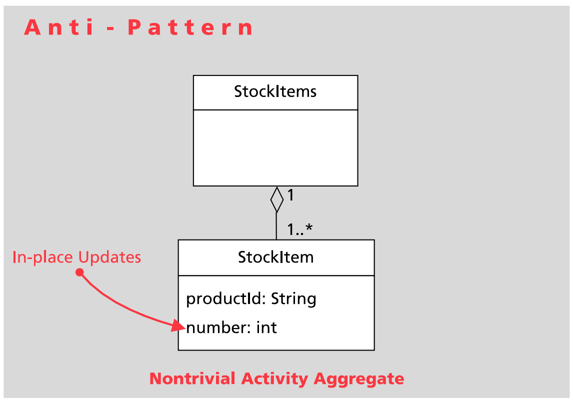
- Does the aggregate contain work results that have been produced in a joint collaboration between different updaters (users or systems)?
- Do multiple users update the aggregate at the same time?
- Is the aggregate rather complex, does it consist of e.g., entities with a lot of attributes or is it composed of a lot of entities and value objects?
- Does the aggregate contain a lot of business operations that are not trivial (a trivial update operation would be a setter method)?

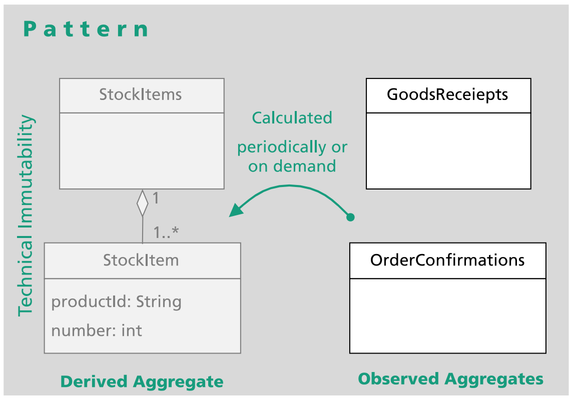
4 Data Model Design Best Practices

In the following, we will describe some data modeling best practices that are based on our taxonomy.

4.1 Trivial Aggregates First

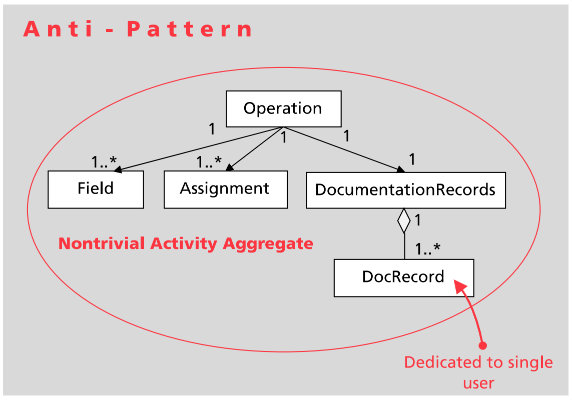
- **Whenever feasible, model aggregates as trivial aggregates**

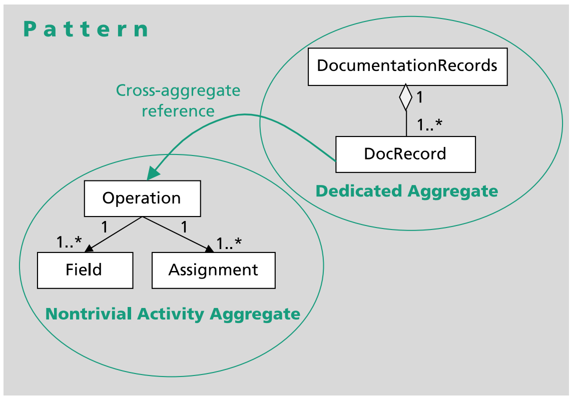




4.2 Separate Dedicated Aggregates

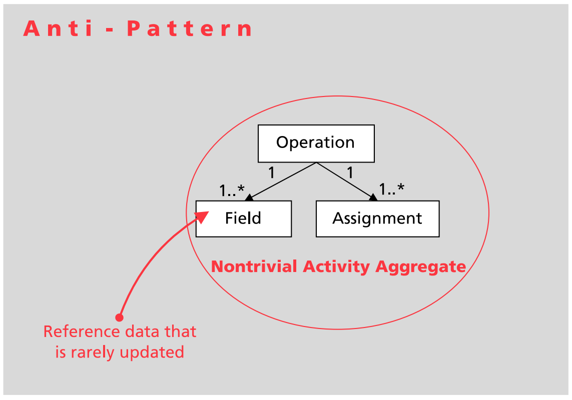
- **Design dedicated data as self-contained aggregate**

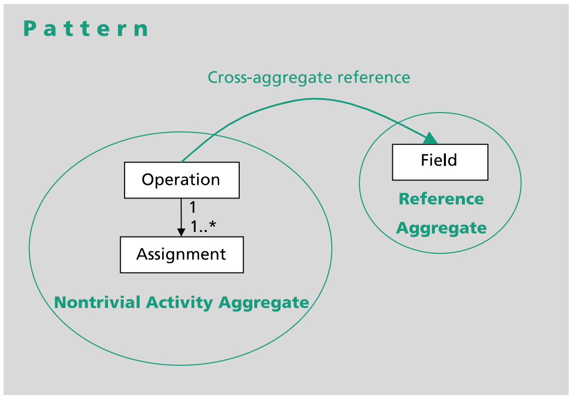




4.3 Separate Different Aggregate Classes

- **Whenever feasible, keep data of different classes in separate aggregates**


Anti - Pattern
Operation
1
1
1..*
1..*
Field
Assignment
Nontrivial Activity Aggregate
Reference data that is rarely updated

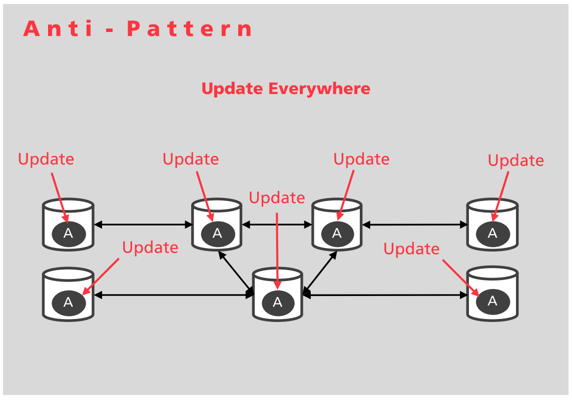


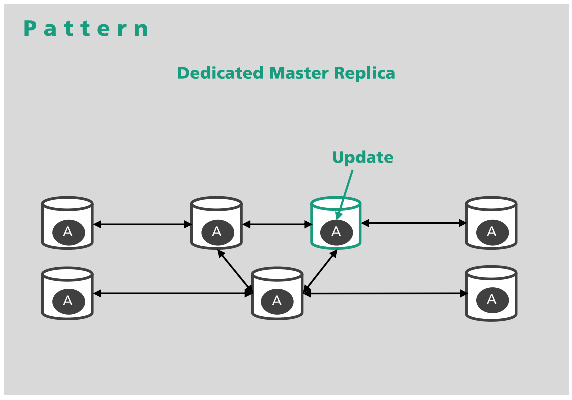
4.4 Idempotent Calculation of Derived Aggregates

The calculation of the state of a derived aggregate should be idempotent & deterministic.

4.5 Apply Master-Slave Replication

- **Consider using Master-Slave-Replication, if transactional (Durability, Serializability, ...) guarantees are required**





5 Cheat Sheet

	Mutable	Concurrent In-Place Updates	Multiple Updater	Update Frequency	Probability of Simultaneous Updates	Probability of Write Conflicts
Immutable Aggregates	✗	✗	✗	✗	✗	✗
Derived Aggregates	✓	✗	✗	✗	✗	✗
Dedicated Aggregates	✓	✓	✗	–	low	low
Reference Aggregates	✓	✓	✓	low	low	low
Activity Aggregates	✓	✓	✓	medium – high	medium – high	medium – high
Collaboration Result Aggregates	✓	✓	✓	medium – very high	high – very high	high – very high

6 References

[1] E. Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software, Addison-Wesley, 2003.

[2] V. Vaughn, Implementing Domain-Driven Design, Addison-Wesley, 2013.

[3] R. C. Martin, Agile software development: principles, patterns, and practices, Prentice Hall, 2002.

[4] C. Baquero and N. Preguiça, "Why Logical Clocks are Easy," *Communications of the ACM,* vol. 59, no. 4, pp. 43-47, 2016.

[5] D. B. Terry, Replicated Data Management for Mobile Computing, Morgan & Claypool Publishers, 2008.

[6] P. Viotti and M. Vukolić, "Consistency in non-transactional distributed storage systems," *ACM Computing Surveys (CSUR),* vol. 49, no. 1, pp. 1-34, 2016.

[7] P. Helland, "Immutability Changes Everything," *ACM Queue - Structured Data,* vol. 13, no. 9, p. 40, 2015.