MicroStream

MicroStream Fundamental Training

Training Course

This training will be recorded!!

Your Trainers

Markus Kett
CEO at MicroStream,
Editor in Chief at JAVAPRO Magazine,
Organizer JCON Conference,
Conference Speaker

Twitter: @MarkusKett
LinkedIn: markuskett
Email: m.kett@microstream.one

Rudy De Busscher
Developer Advocate,
Trainer & Conference Speaker

Twitter: @rdebusscher
Email: r.debusscher@microstream.one

Day 1

Topics

- Day 1
  - Markus Kett MicroStream introduction
  - Performance demo
  - MicroStream quick demo
  - Creating a proper data-model
  - Step by step configuration „MicroStream“
  - Configuration in detail
  - Getting started with runtimes
  - CRUD – Project
  - Exercise
- Day 2
  - Review the exercise
  - Java 8 Streams
  - LazyLoading
  - LegacyTypeMapping
  - Backup strategies
  - Project development workflow (BestPractises)
  - TBD
    - (Attendee Requests)
    - Write data to Azure Storage
    - Write data to PostgreSQL

Introduction to MicroStream

By Markus

Attendees

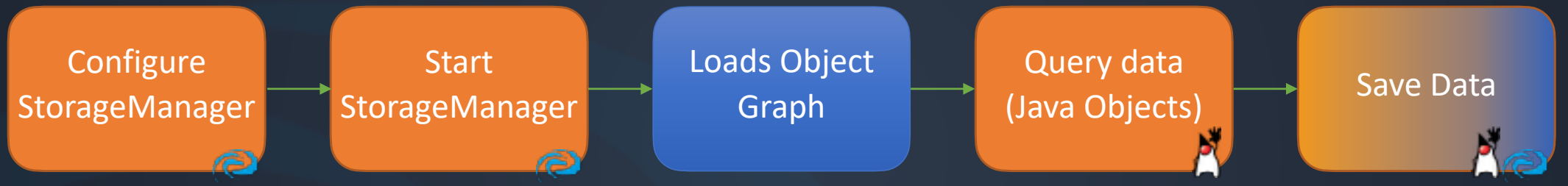
Material

- Github repository : https://github.com/rdebusscher/MicroStream-Basic-training
- Java 11
- Maven
- IDE

- Tuesday 9:00 -> 16:00. (Break between 12:00 and 13:00)
- Wednesday 9:00 -> 16:00. (Break between 12:00 and 13:00)

- Questions
  - Ask as much questions as possible whenever you want

Overview



Developer

MicroStream “magic”

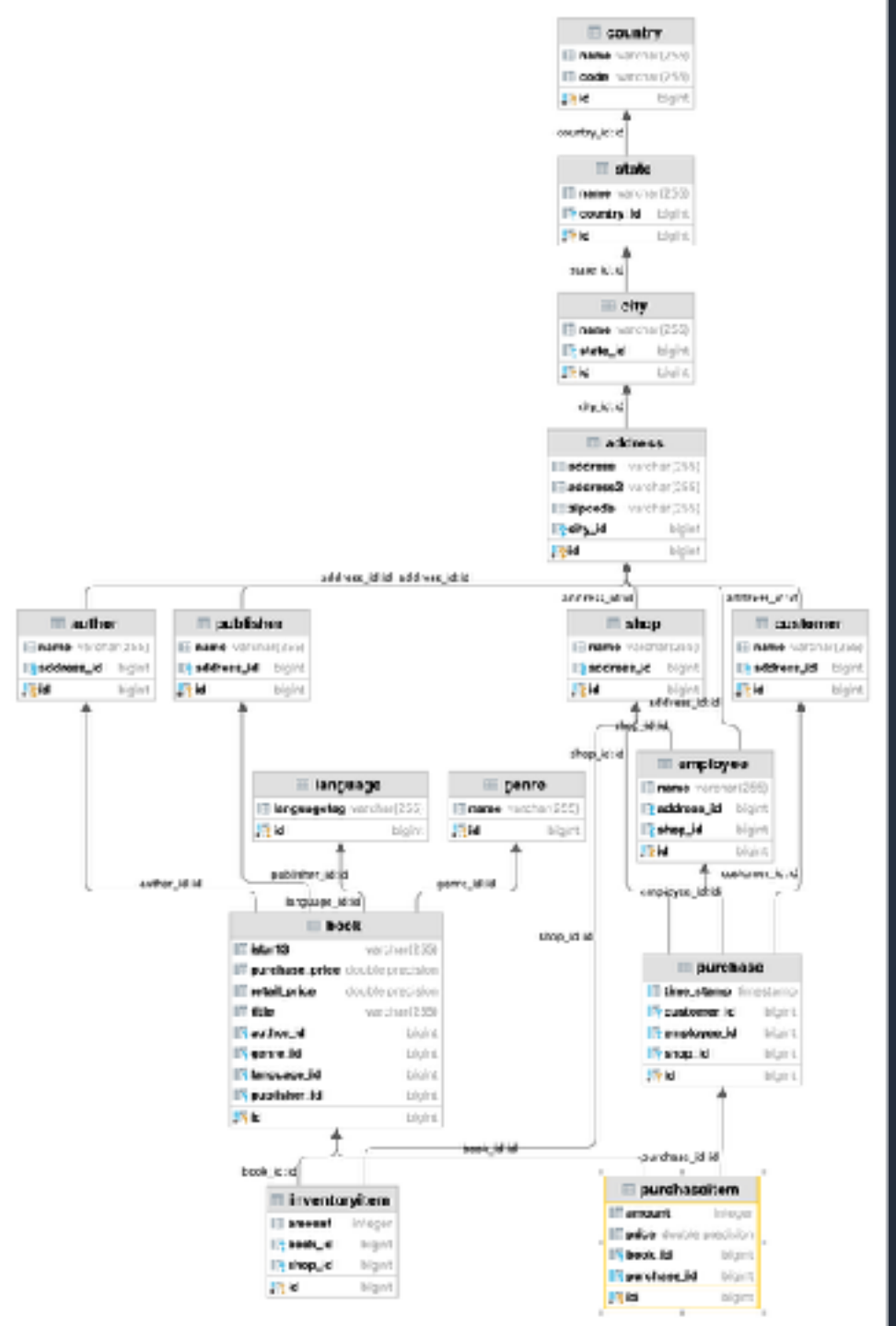
Pure Java

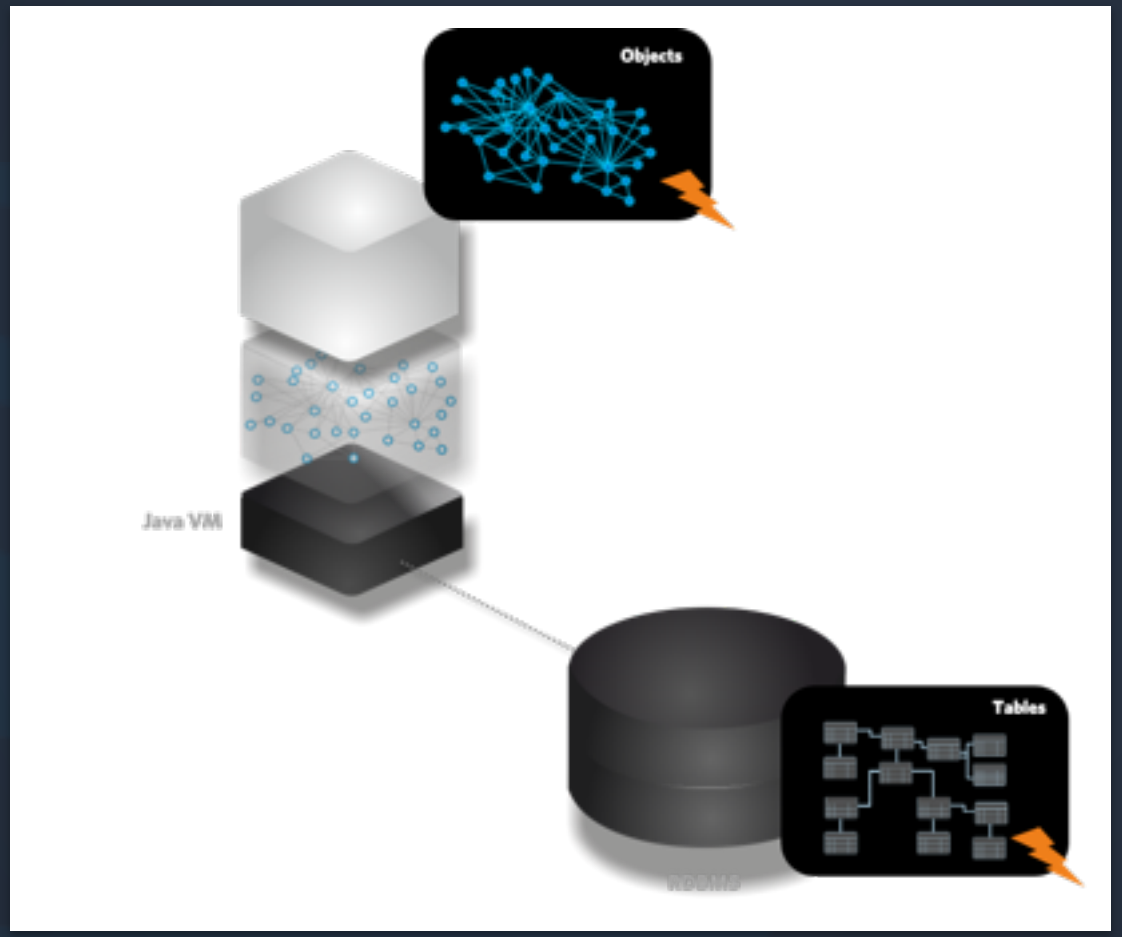
MicroStream API

Performance Demo

Or why you might drop external data storage as primary source

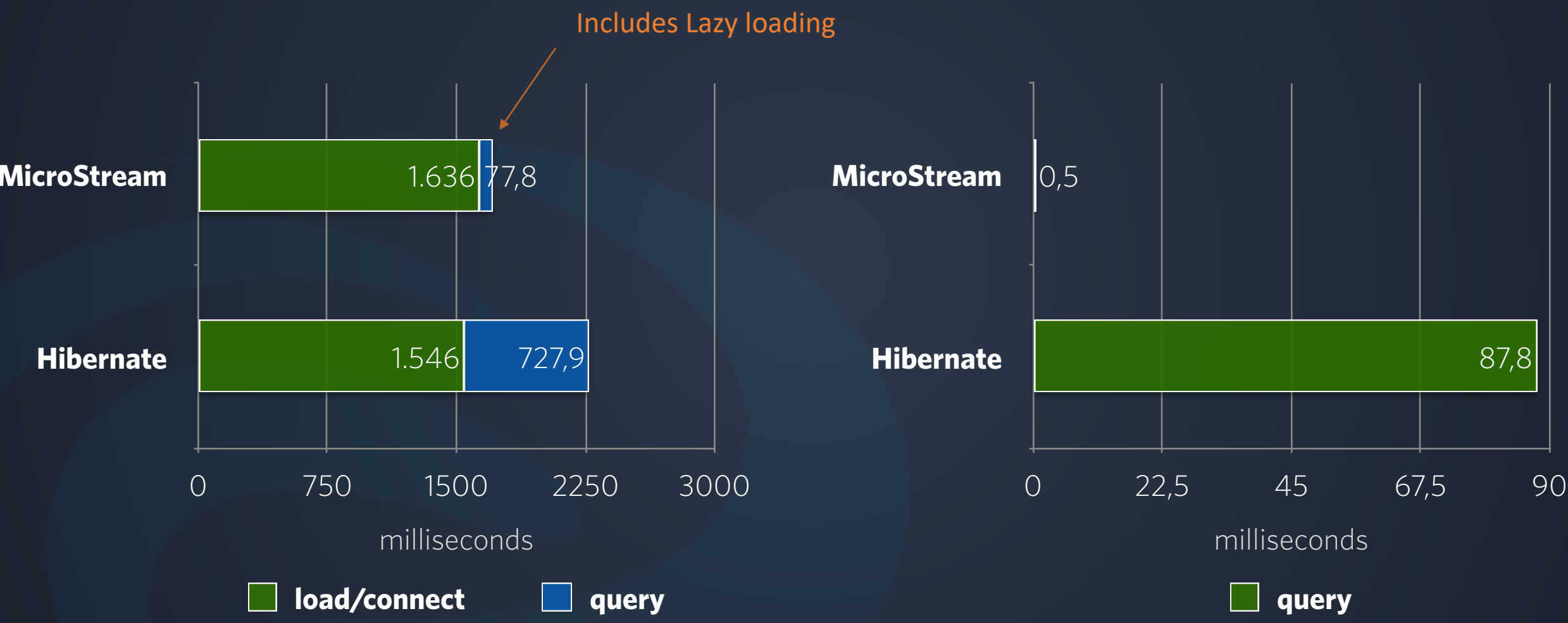
Data is Everywhere


country
state
city
address
author
publisher
shop
customer
language
genre
employee
book
purchase
inventoryitem
purchaseitem


Objects
Java VM
Tables

Demo

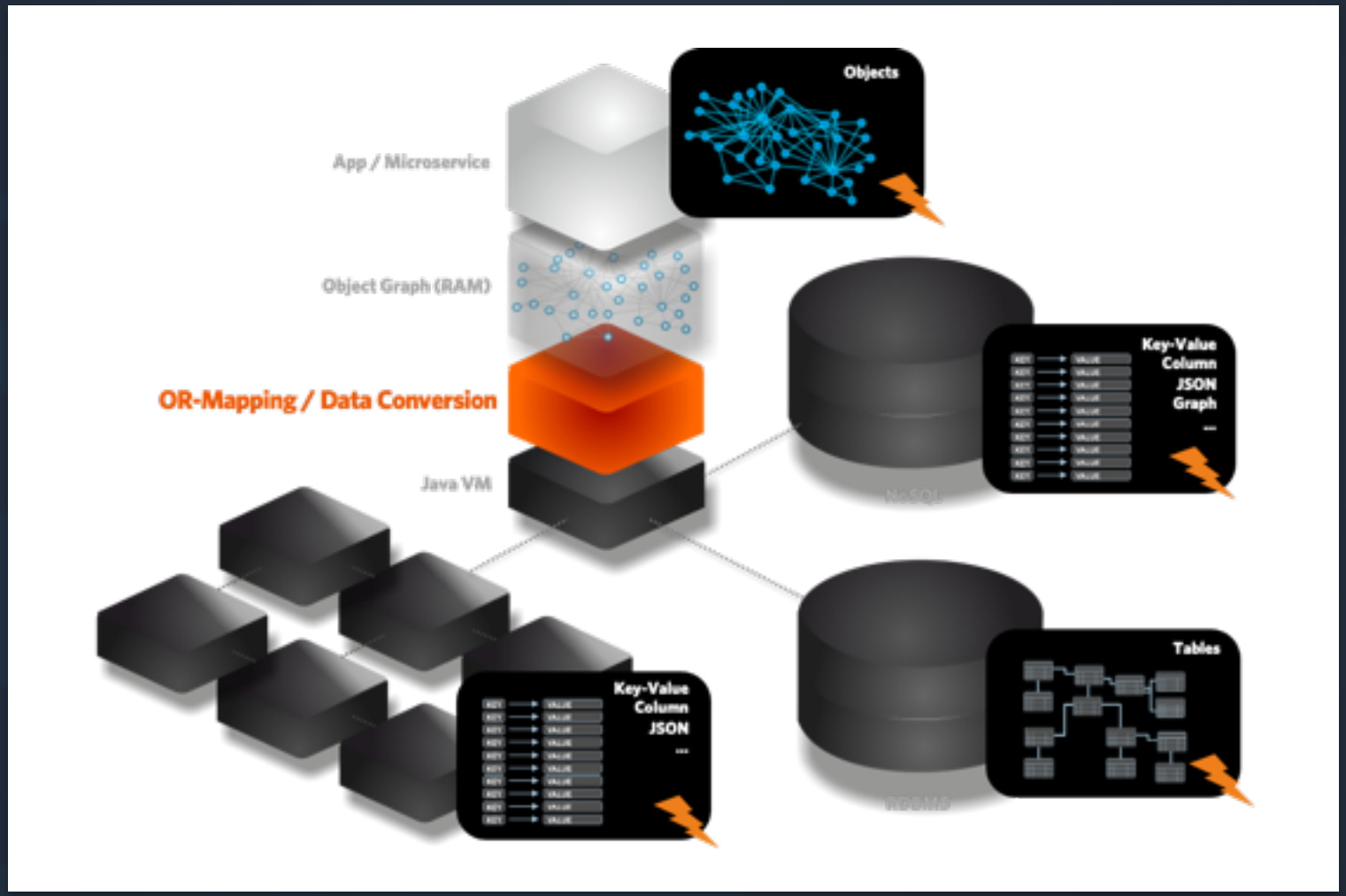
Performance comparison



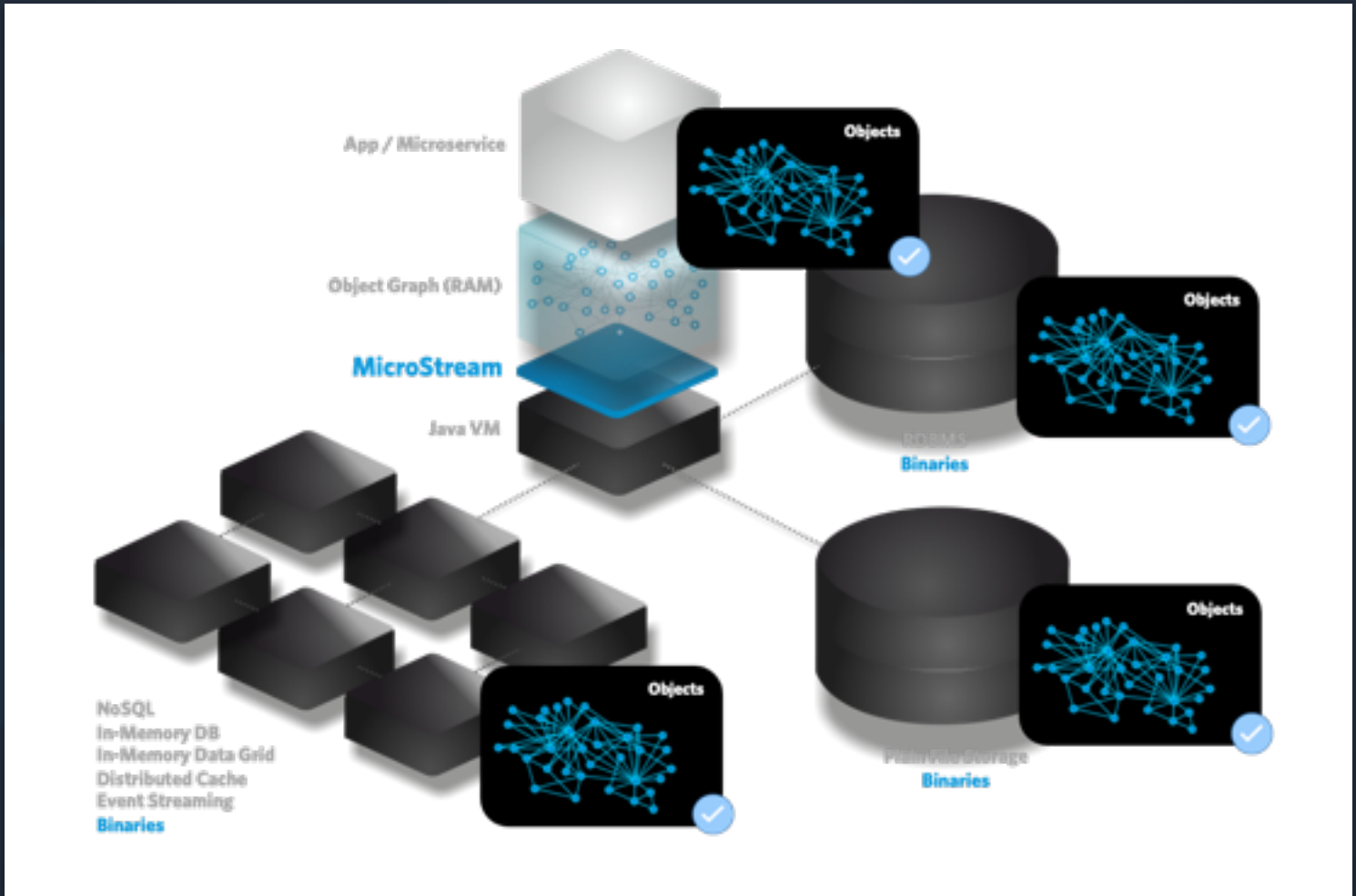
Latency numbers

```
Latency Comparison Numbers (~2012)
----------------------------------
L1 cache reference                           0.5 ns
Branch mispredict                            5   ns
L2 cache reference                           7   ns                      14x L1 cache
Mutex lock/unlock                           25   ns
Main memory reference                      100   ns                      20x L2 cache, 200x L1 cache
Compress 1K bytes with Zippy             3,000   ns        3 us
Send 1K bytes over 1 Gbps network       10,000   ns       10 us
Read 4K randomly from SSD*             150,000   ns      150 us           ~1GB/sec SSD
Read 1 MB sequentially from memory     250,000   ns      250 us
Round trip within same datacenter      500,000   ns      500 us
Read 1 MB sequentially from SSD*     1,000,000   ns    1,000 us    1 ms  ~1GB/sec SSD, 4X memory
Disk seek                           10,000,000   ns   10,000 us   10 ms  20x datacenter roundtrip
Read 1 MB sequentially from disk    20,000,000   ns   20,000 us   20 ms  80x memory, 20X SSD
Send packet CA->Netherlands->CA    150,000,000   ns  150,000 us  150 ms
```

Data Storage (Traditional)



MicroStream Persistence




TIME FOR A
BREAK

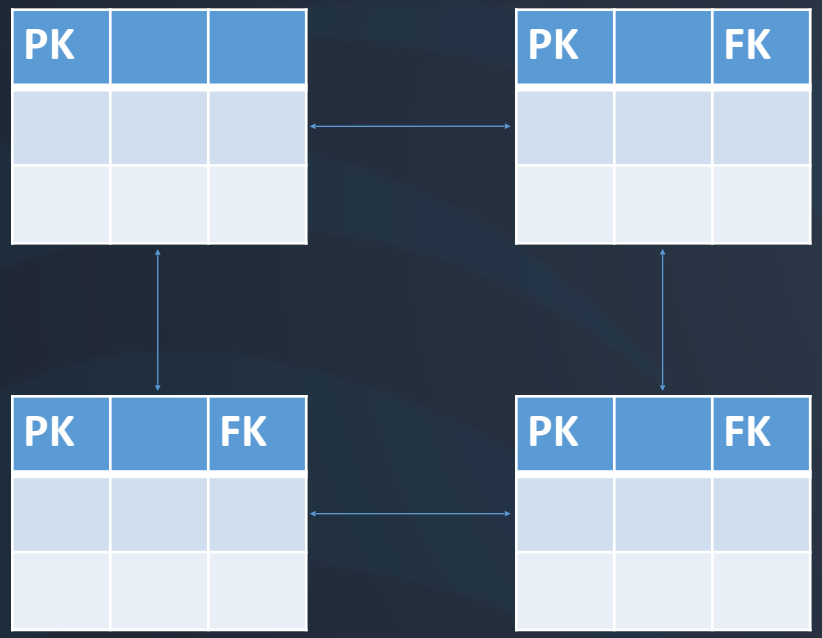
Proper Data Model

Topics

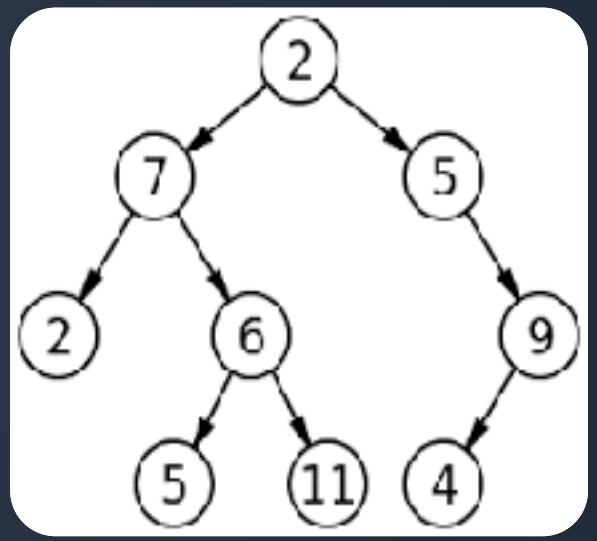
- Differences between
  - Relational model
  - Graph model
- Opportunities to design a graph
- How to bi-directional
- How does the graph influence the UI
- How can each ‘entity’ be accessed
- Frequency of data access
  - Historical data

Relational database vs a graph

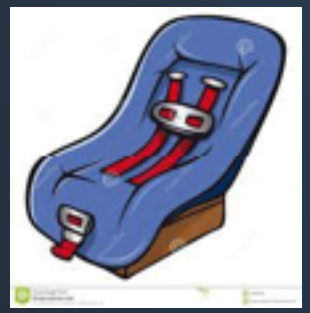
Relational model



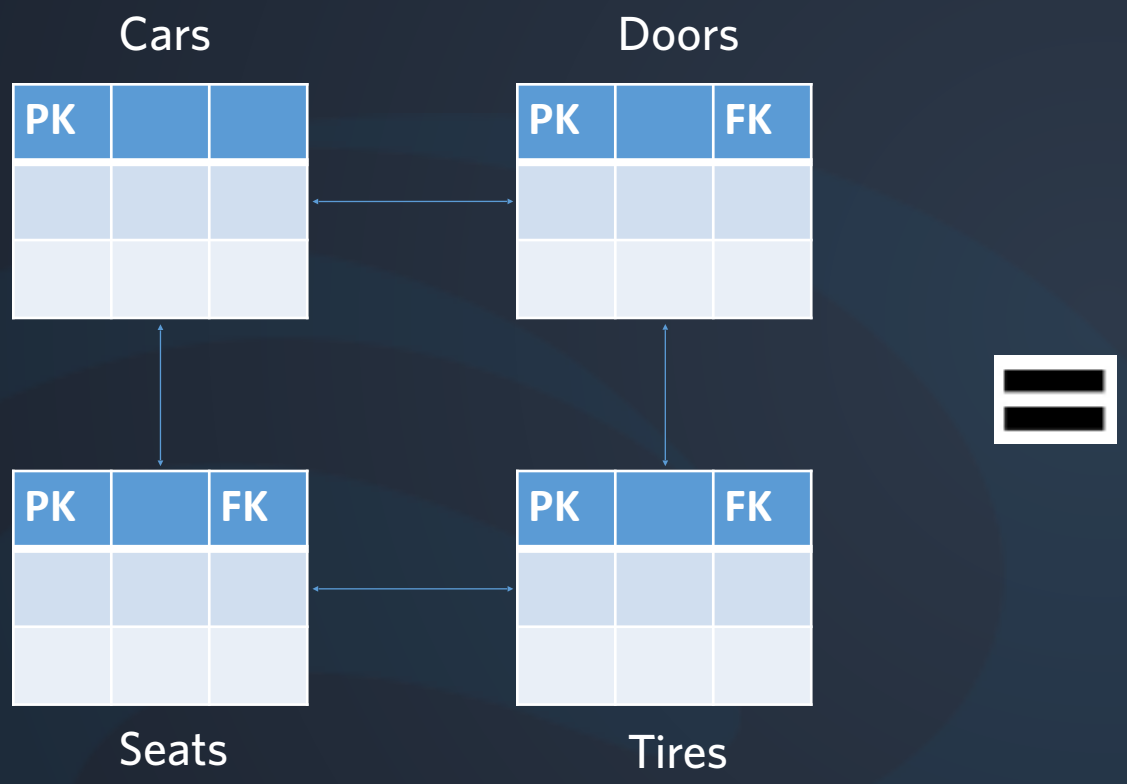
Object Graph

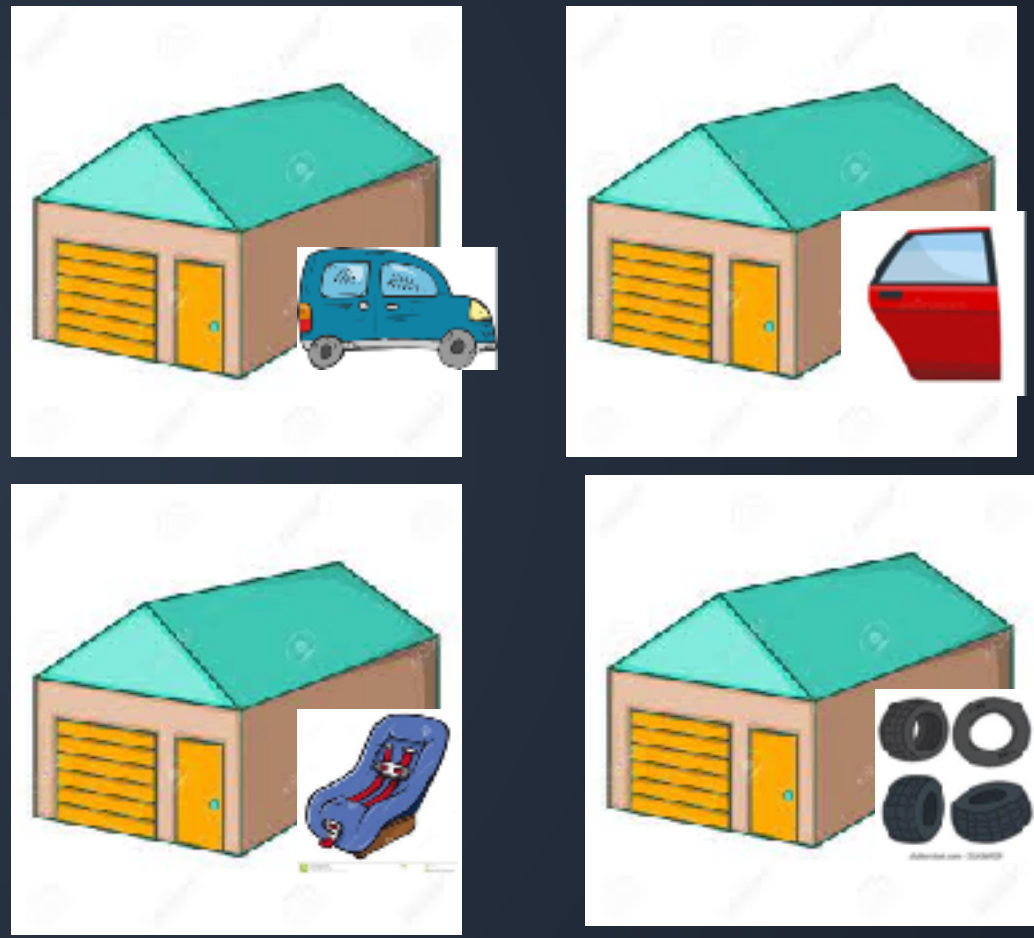


Relational database vs a graph

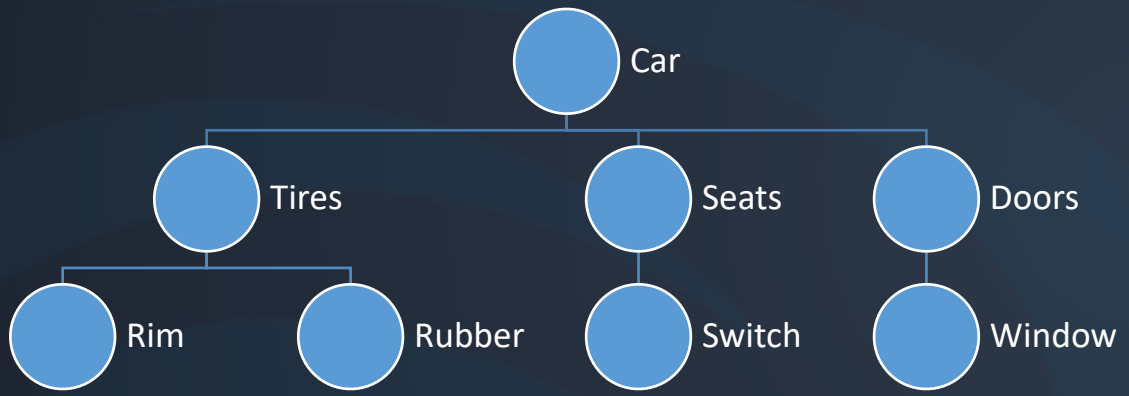


Relational model vs real world



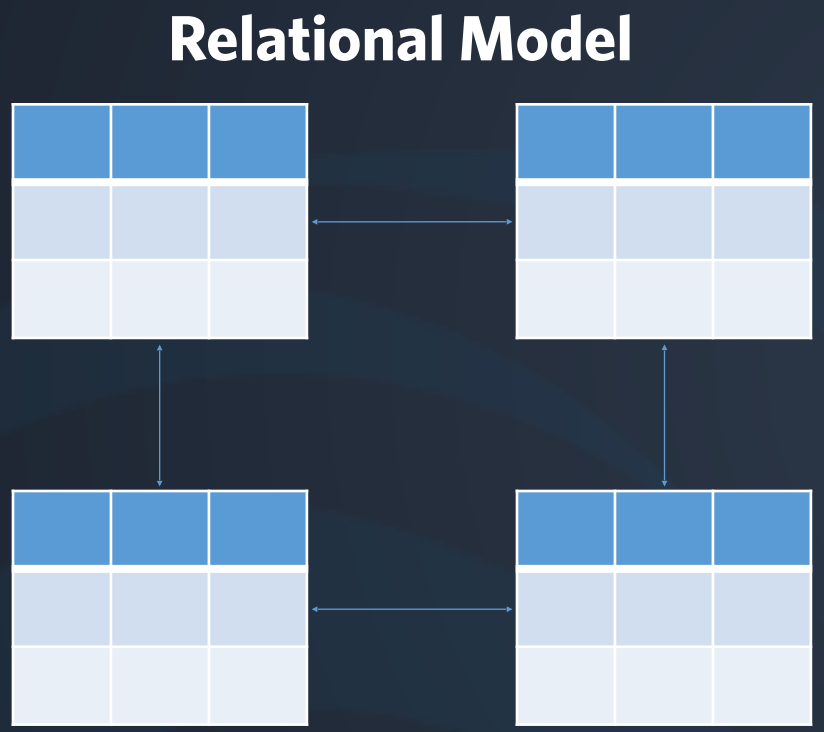


Object graph vs real world



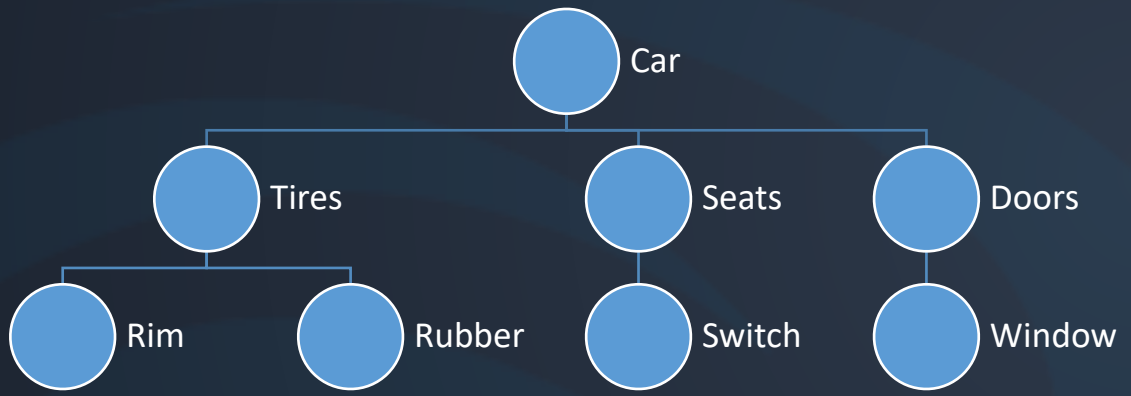
=

Relational database vs a graph



- Data within a table are incomplete
- Informations about relations are absolutely necessary (Code)
- Informations about relations are unnecessary (Database)
- Informations about relation are missing
- Constraints defined in the database are not helpful for the developer

Relational database vs a graph



- A graph looks like the real world
- Relations are directly defined within the node
- A Java Object „Car“ represents a single Car with all informations in it.

Considerations regarding the data model

Considerations regarding the data model

Car

N — 1

Brand

Brand
- String name - List<Car> cars

Car
- String name - BigDecimal price - ...

or

Car
- String name - BigDecimal price - Brand brand - ...

Brand
- String name - ...

Car
- String name - BigDecimal price - Brand brand - ...

Brand
- String name - List<Car> cars

Considerations regarding the data model

- Is it possible to have a car without a brand? If Yes...

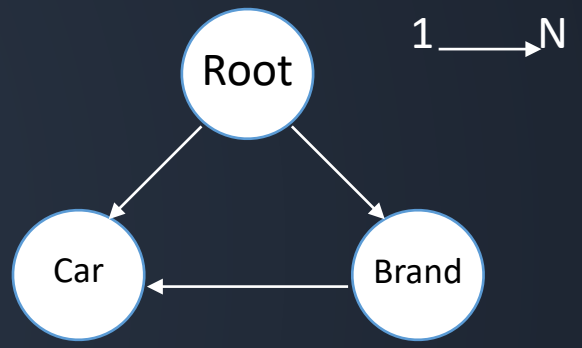
Car
- String name - BigDecimal price - ...

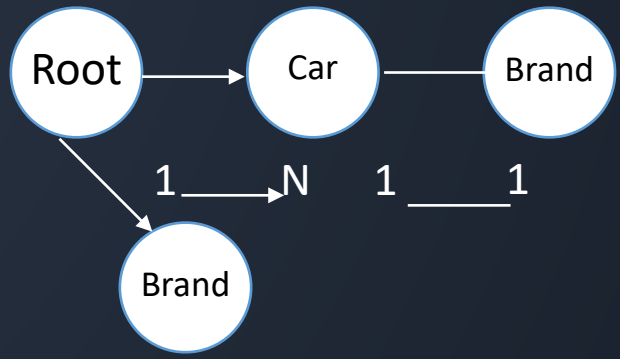
Brand
- String name - List< Car > cars

Car
- String name - BigDecimal price - Brand brand - ...

Brand
- String name - ...

BUT





Considerations regarding the data model

- Influence of the data model to the later UI. Brand or Category is no longer available as an attribute. How can this work in my forms and code.
  - Maybe some framework features of the UI are no longer usable.
  - Maybe the code to store a Product is totally different.
  - If i have only the „Brand“ how to get all Products of a brand?

Considerations regarding the data model

- Influence of the data model to the later performance...

```
Optional<Category> category
        = Root.getCategories()
                .stream()
                .filter(o -> o.getProducts().contains(product))
                .findFirst();
```

Name	Preis	Kategorie
...	...	...
...	...	...
...	...	...
...	...	...

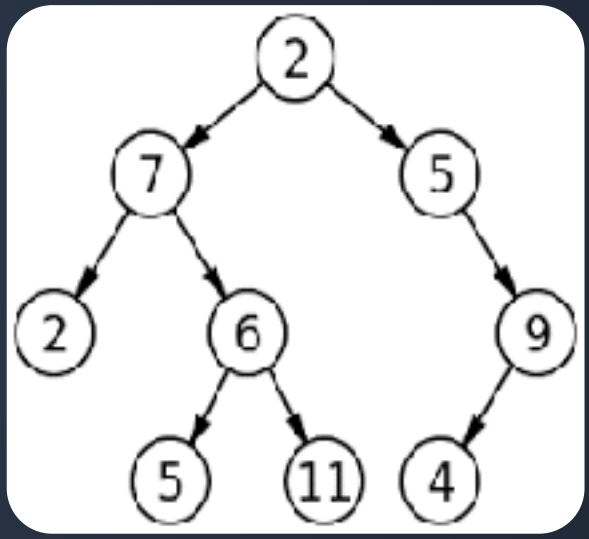
... 500 000 Datensätze

Considerations regarding the data model - Bi-directional

1. Bi-directional is kind of a redundancy. You are in charge to keep it consistent.
2. Does it needs more memory on the RAM? Yes, not for the Data itself because it is a reference but the reference itself needs RAM.
3. A huge amount of references can lead to a huge amount of RAM. Keep in mind MicroStream is an in memory database..

Accessing data

- Object graphs need to be searched from the root
- How to get to a specific instance fast?
- How to get list of <11>
  - What if there are millions of items <6>



Indexing

- **Database**

- Index
- *where <indexfield> = <value>*

- Multiple indexes

- **Pure Java**

- Map (HashMap)
- *map.get(<value>)*

- Map<String, Book> *(isbn number)*
- Map<Author, List<Book>>

Multiple references

- Use Map like structure to create fast access paths
- Multiple indexes to same instance possible
  - Only overhead of the reference
  - Additional statement(s) to keep in-sync
  - Balance with performance trade-off

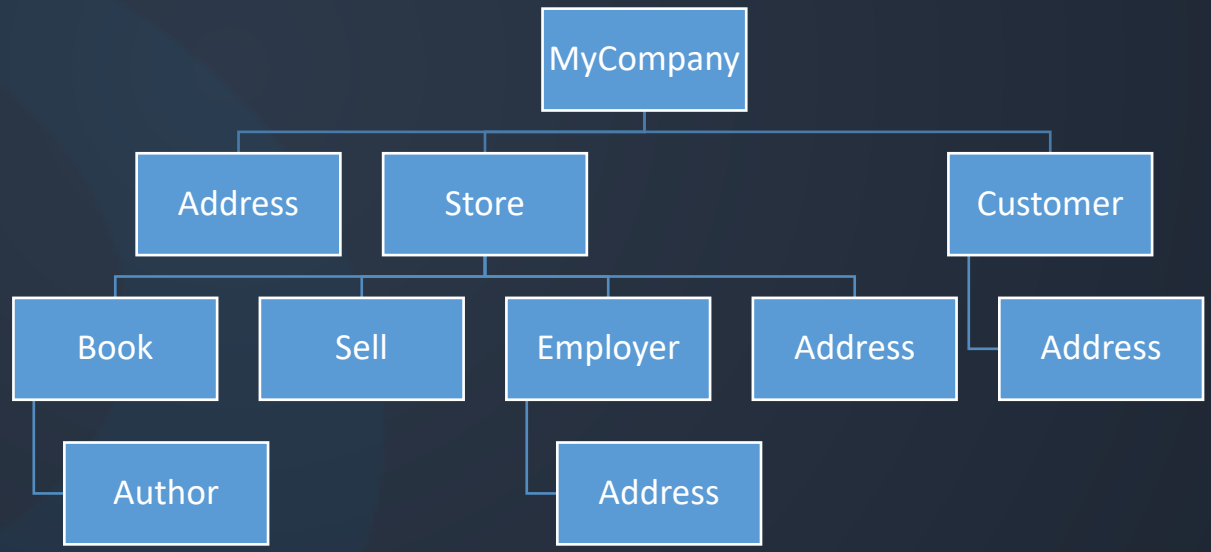
Historical data

- In general : Not frequently used data
- Don't keep everything in 1 giant List
  - Apply indexing to group similar data
  - By year, year-month combination, ...
- Faster access
- Lazily loading (see further-on in training)

Example

Imagine: We are a book selling company. With multiple bookstores. We are selling thousands of books a day to our customers. **What most important notes / tables do we need and how the relations would look like.**

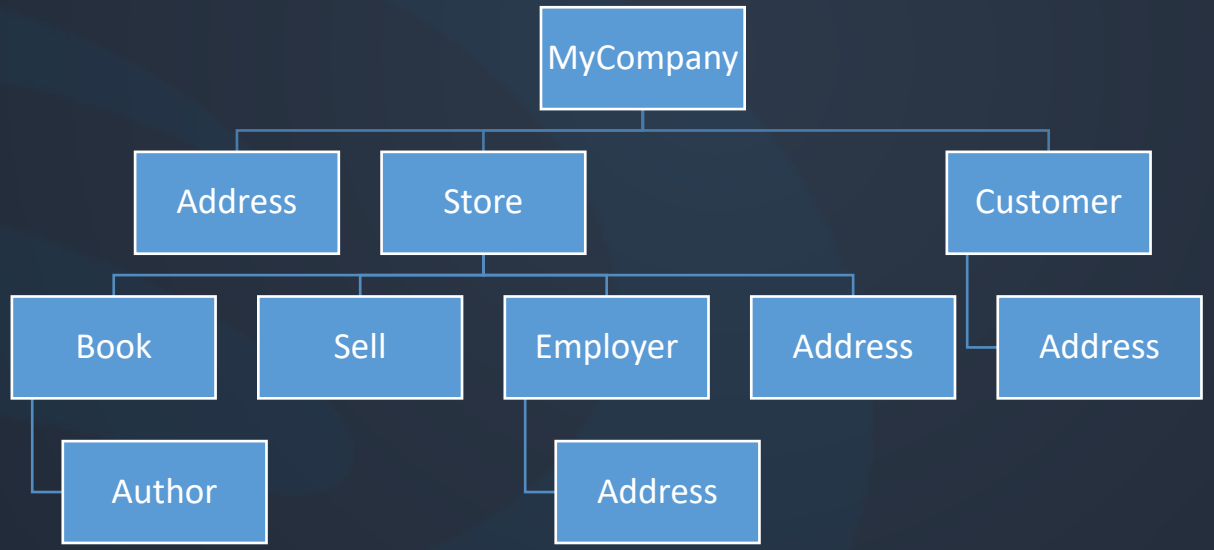
1. Our Company
2. Books
3. Authors
4. Stores
5. Customers
6. Addresses
7. Sells
8. Employees



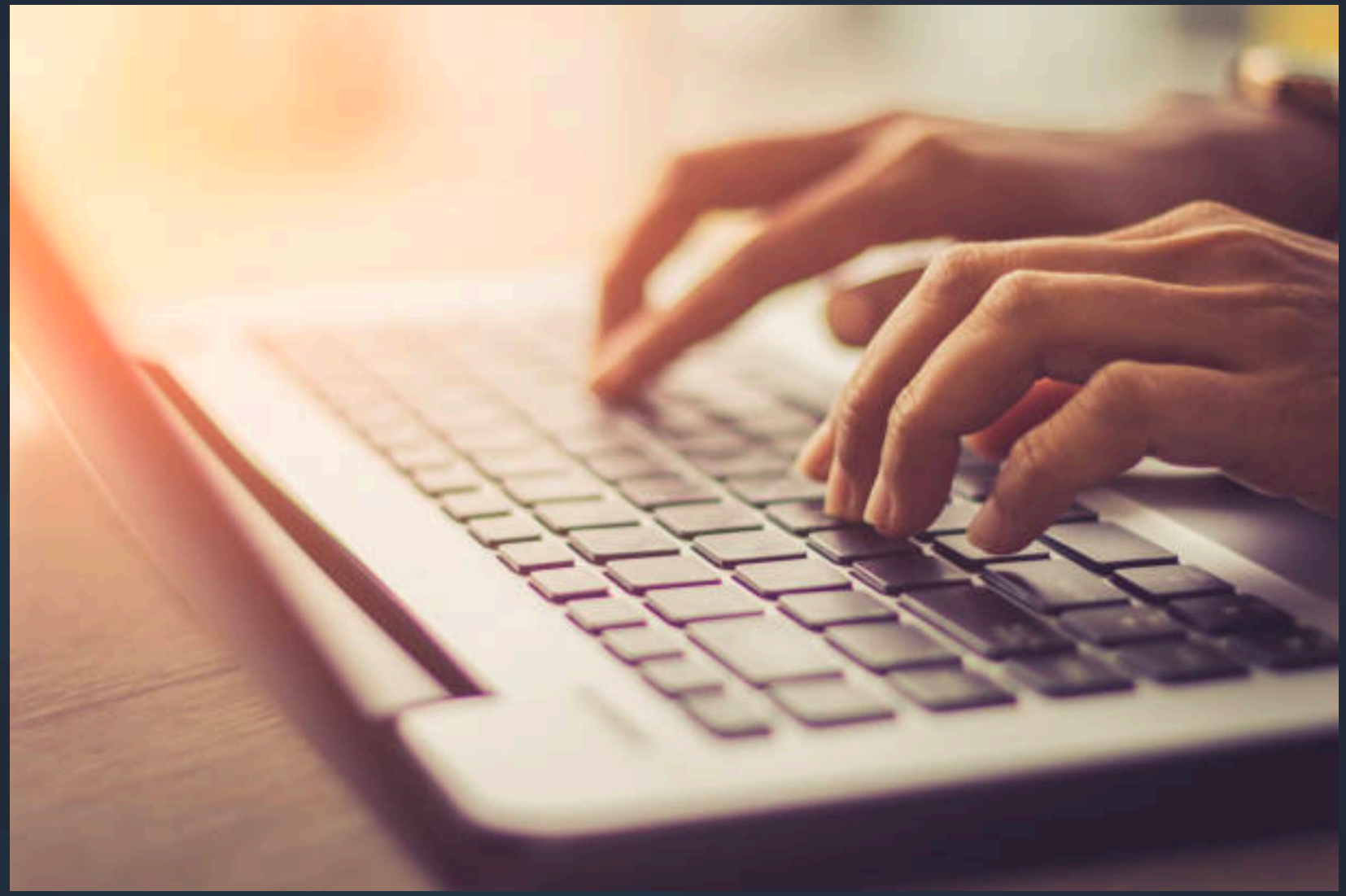
Example

That was easy right? But ...

1. **What about if we need a ComboBox of authors in the app?**
2. **What about if a book is sold out in every store? Where to find the book now?**



Some hands-on



Exercise Templates

Exercise templates

- See Github Repo

- For this training
- Not using dedicated implementations

- Singleton pattern
  - One-time configuration - initialisation

Demo


TIME FOR LUNCH

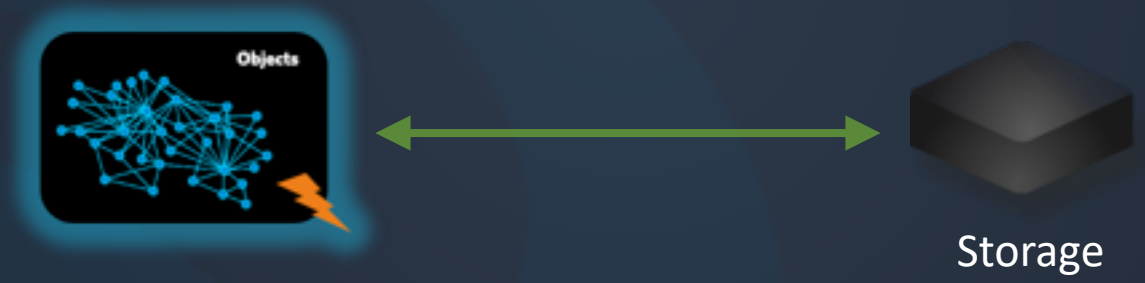
Configuration of Storage Manager

“The most difficult task”

But can be done in a single statement

Minimal Configuration

- Provide the Object root and indicate where there storage is located



Demo

AFS

- Abstract File System

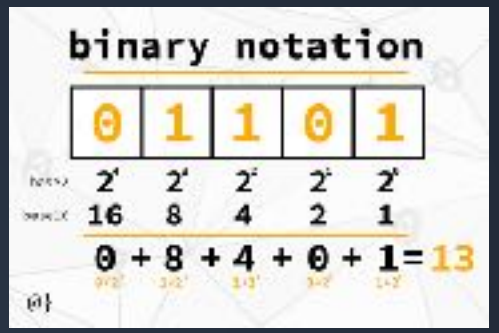
NIOFile

SQL

Blob

Channel Count

- Power of 2. (1, 2, 4, 8, 16, ...)

System.identityHashCode() → 

→ Channel Number

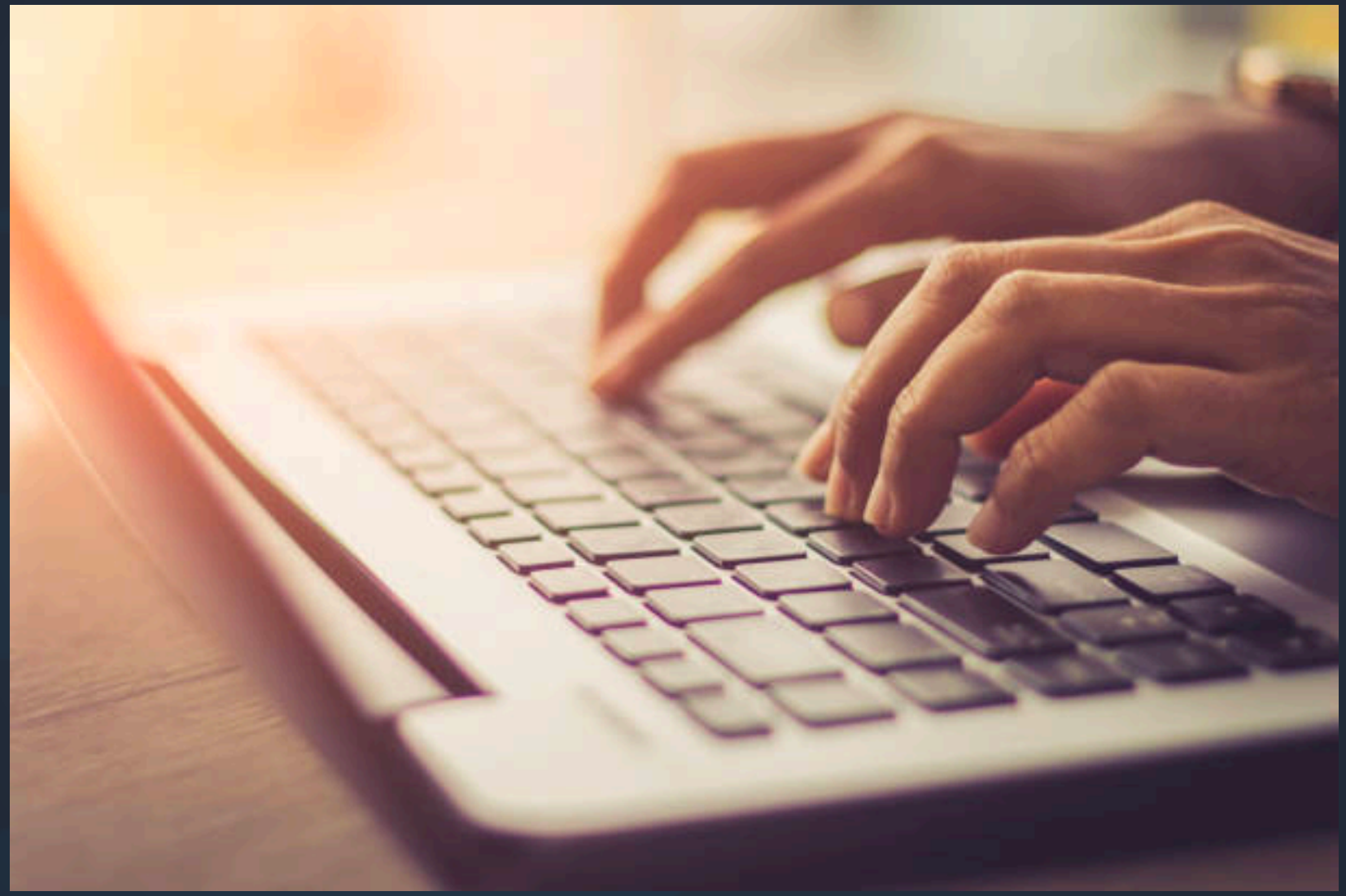
- Number of Threads performing "IO"
- When to set?
  - Large initial data set to load
  - Huge Object change rate

Other config aspects

- Classloader
  - When runtime has multiple/hierarchical class loaders
    - MicroStream code within other ClassLoader as application classes
    - ```
      storageFoundation.onConnectionFoundation(cf ->
          cf.setClassLoaderProvider(ClassLoaderProvider.New(
              Thread.currentThread().getContextClassLoader())));
      ```

- Specialised Handler
  - A class can have a dedicated handler to handle binary conversion
  - Are in dedicated MicroStream dependencies
  - ```
    storageFoundation.onConnectionFoundation(
        BinaryHandlersJDK17::registerJDK17TypeHandlers);
    ```

Some hands-on



Interact with Data

Store = external for safe process restart

(Not primary source)

Load

- When Storage Manager created, Object graph is loaded from storage location

- No simple API available to reload.
  - reload of (sub) graph is possible -> advanced

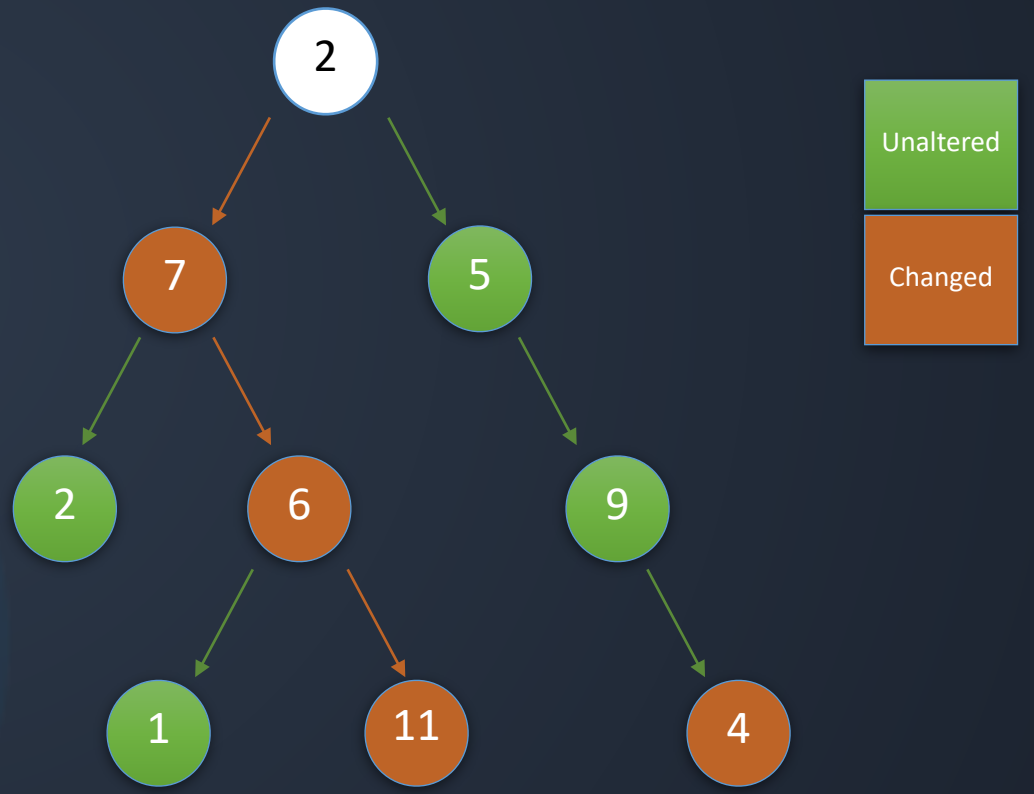
- Read
  - Use Java API
  - Any java statement can be used.
  - Stream API

Storing data

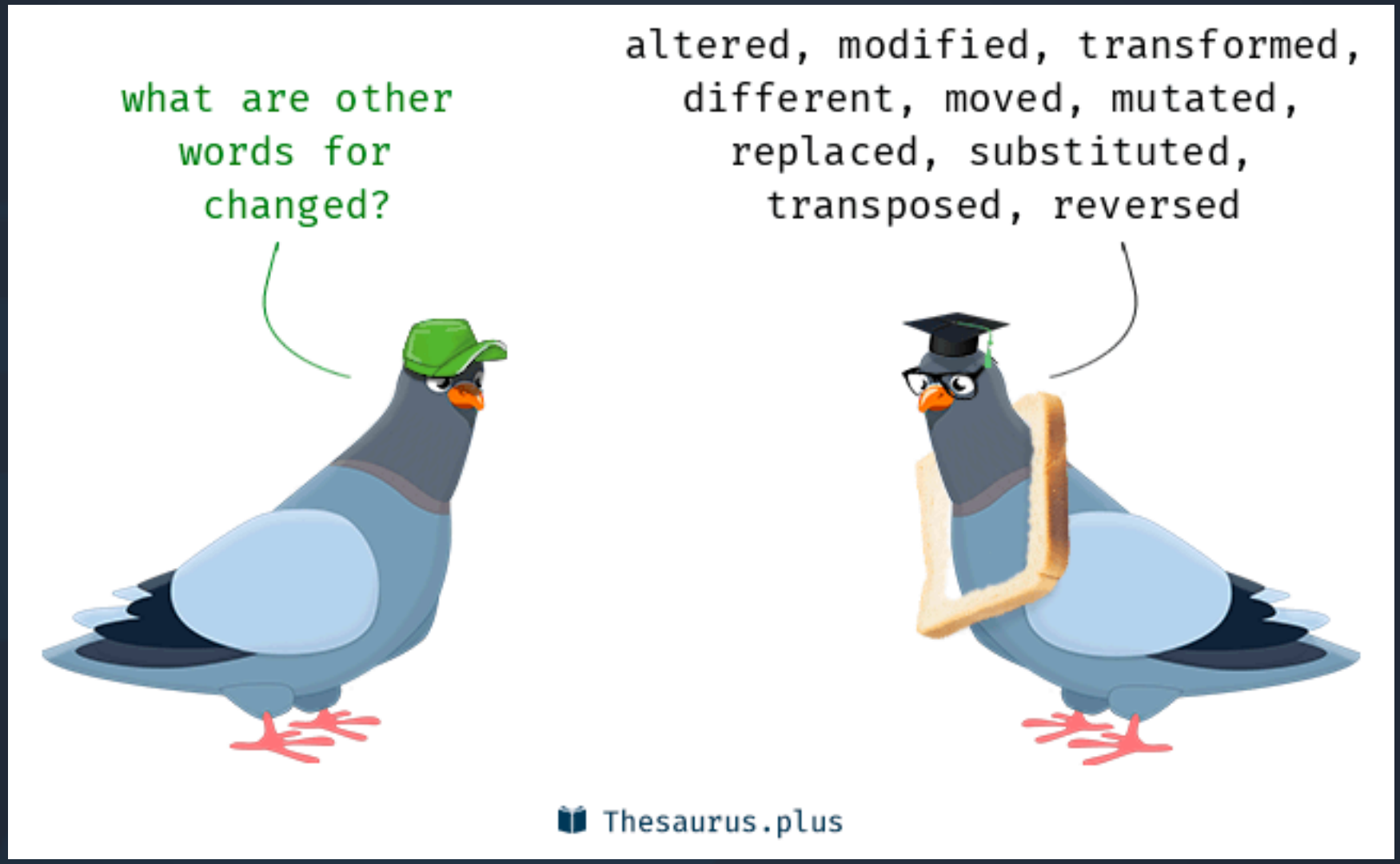
- storeRoot()
- store(o)

- Default
  - LazyStorer
  - Traverse tree/graph
  - Stops when *unchanged* pointer.
- Eager Storer
  - Stores all objects

- Keeps track of all state (pointers)



Store what is changed!



Demo


TIME FOR A
BREAK

Storer

- storageManager.createEagerStorer()
- storageManager.createLazyStorer()

- storer.store(o)
- storer.commit() !!!
  - storageManager.store(o) combines *store()* and *commit()*

- Multiple changes send to external storage

How to store()

- Add or remove item from collection -> store(collection)
- New value for key in Map -> store(map)
- Update property in POJO -> store(pojo)

- Help MicroStream algorithm
  - Use immutable objects
  - Mainly useful for collections.

- Before commit off-heap native memory used.
  - Important for *containerised* deployments.


Freitag
Friday
18
Vendredi
Viernes
Exercise!
Samstag
Saturday
19

Day 2

Recapitulation

- Why MicroStream?
  - External data storage is slow, requires mapping, tuning, not productive
  - Query is much faster but also writes.
- Data-model
  - Relational datamodel vs Object graph
  - Relations - how; access - different views
- Configuration
  - AFS
  - Different options
- CRUD
  - Store what is changed
  - Lazy vs Eager Storer

Topics

- Day 1
  - Markus Kett MicroStream introduction
  - Performance demo
  - MicroStream quick demo
  - Getting started with runtimes
  - Creating a proper data-model
  - Step by step configuration „MicroStream“
  - Configuration in detail
  - CRUD – Project
  - Exercise
- Day 2
  - Review the exercise
  - Java 8 Streams
  - LazyLoading
  - LegacyTypeMapping
  - Backup strategies
  - Project development workflow (BestPractises)
  - TBD
    - (Attendee Requests)
    - Write data to Azure Storage
    - Write data to PostgreSQL

Exercise discussion time

MicroStream

Omitting Data from Graph

Transient data

- All data / properties are stored
- Except if property is *marked* as **transient.**

- Define custom *PersistenceFieldEvaluator*

Java 8 Streams

MicroStream Query language

What?

- Not a data structure
- Series of processing steps
- Terminal operation to create the stream result.
- Only terminal operation result starts the entire pipeline

Stream elements

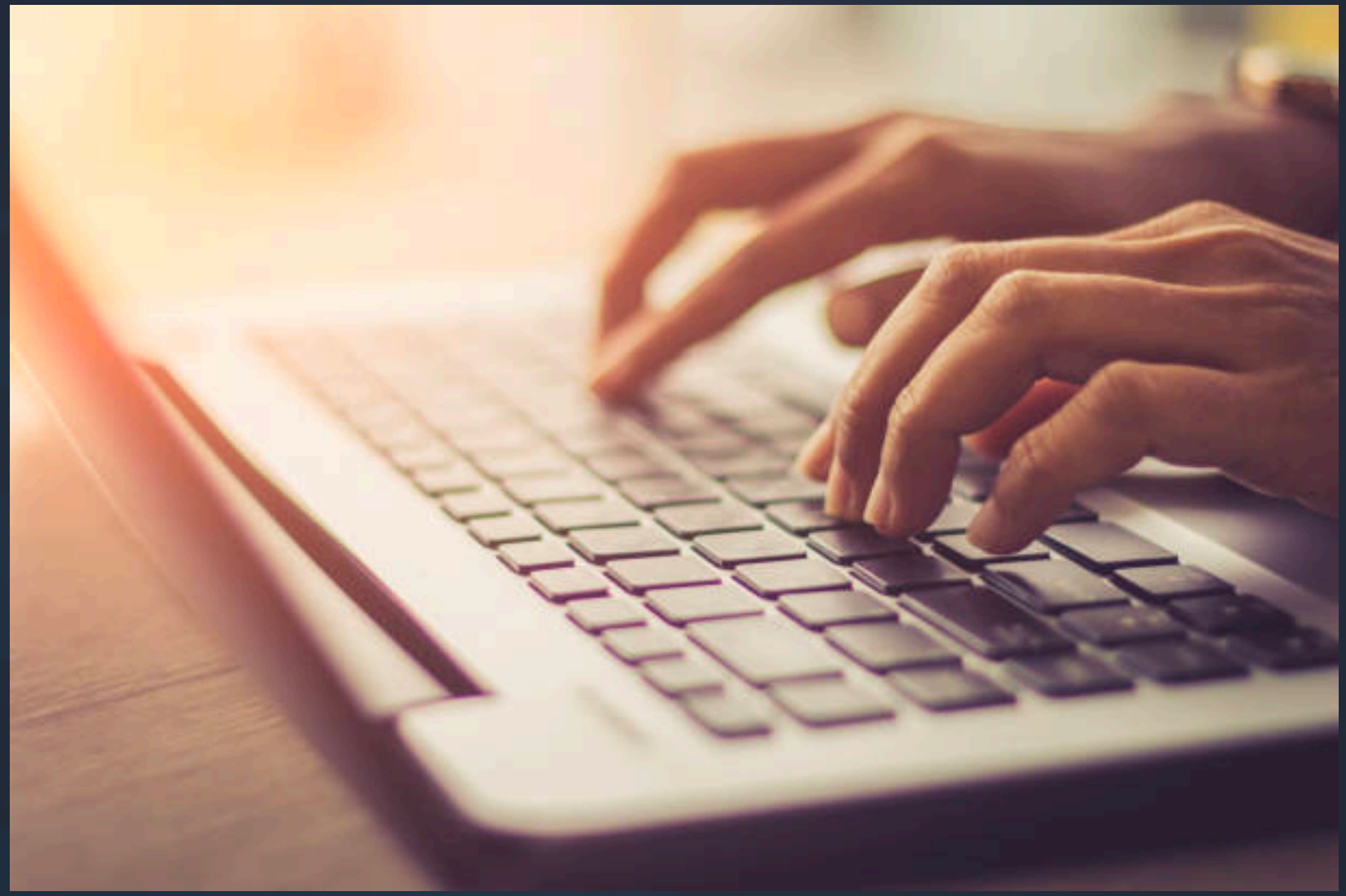
- Intermediate steps
  - Filter
  - Map
  - Sort
- Terminal
  - Consume
  - Collect
  - Reduce / Aggregate

- Serial or parallel processing possible
  - Parallel only in a few cases more efficient.

Java Stream and MicroStream

- The query language for collections.

Some hands-on



Basic Streams (Examples)

- Processing data
- No return values. -> Consume the stream

	Code
IntStream	IntStream.of(1, 2, 3, 4, 5, 6, 7, 8, 9).forEach(i -> System.out.println(i)); IntStream.range(5, 10).forEach(i -> System.out.println(i));
ArrayStream	Integer[] nums = {1, 2, 3, 4, 5, 6, 7, 8, 9}; Stream.of(nums).forEach(i -> System.out.println(i)); Arrays.stream(nums).forEach(System.out::println);
CollectionStream	DB.root.getBooks().forEach(b -> System.out.println(b.getName()));

Streams (Collectors)

	Code
„ToArray“ Collector	DB.root.getBooks().stream().toArray(Book[]::new);
„ToList“ Collector	DB.root.getBooks().stream().collect(Collectors.toList());
„ToSet“ Collector	DB.root.getBooks().stream().collect(Collectors.toSet());
„StringJoin“ Collector	List<String> petList = Arrays.asList("Cat", "Dog", "Mouse", "Bird"); return petList.stream().collect(Collectors.joining(","));

	Code
„findAny“ SingleObjects	return DB.root.getBooks().stream().findAny().get();
„findFirst“ SingleObject	return DB.root.getBooks().stream().findFirst().get();

Streams (Filter)

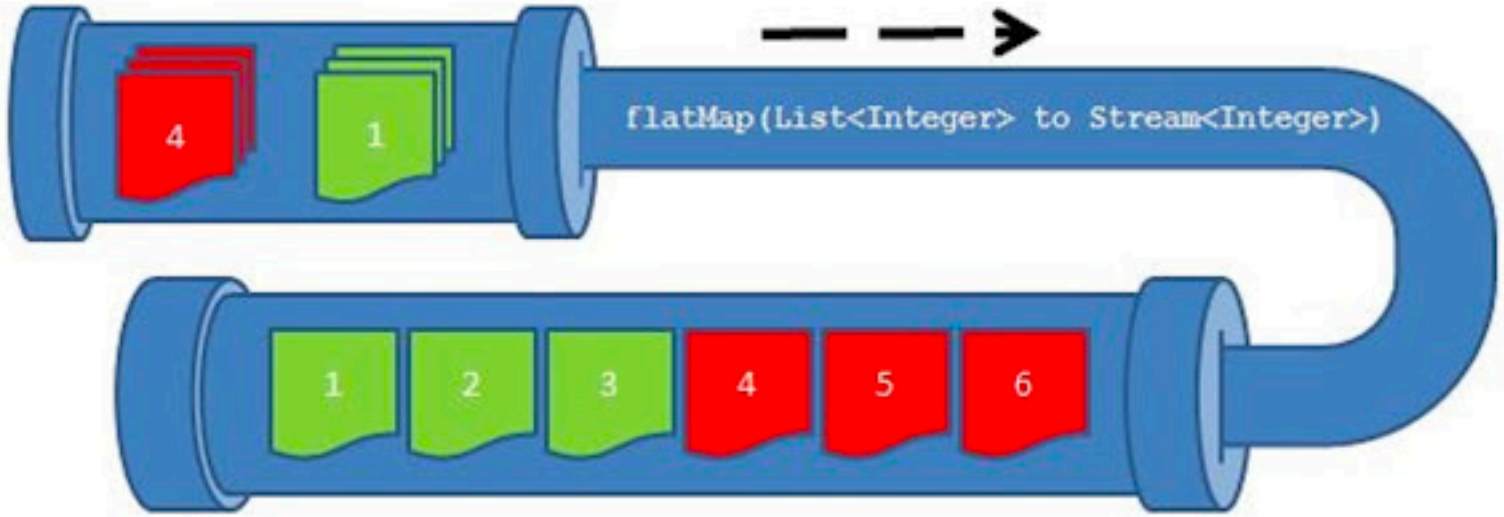
Filter	Code
StartsWith	**Return** DB.**root**.**getBooks().stream()** **.filter(b -> b.getName().startsWith("A"))** **.collect(Collectors.toList());**
Equals	**DB.root.getBooks().stream().findFirst().get();**
Equals Pedicate	Predicate<Book> namePredicate = b -> b.getName().equals(“Bangkok Dangerous”); Predicate<Book> isbnPredicate = b -> b.getIsbn().equals("12345"); Predicate<Book> nameAndISBNPredicate = namePredicate.and(isbnPredicate); **DB.root.getBooks().stream()** **.filter(nameAndISBNPredicate)** **.collect(**Collectors.toList());**

Streams (Map vs. FlatMap)

- Map stands for: Map the result to a different result type
- FlatMap stands for: Merge/join each stream to a single stream.

	Code
Map	List<Author> collect = DB.root.getBooks().stream() .map(c -> c.getAuthor()) .collect(Collectors.toList());
FlatMap	List<Address> addresses = getAllAuthors().stream() .flatMap(a -> a.getAddresses().stream()) .collect(Collectors.toList());

- The `flatMap` operation




```
List<Integer> together = Stream.of(asList(1, 2, 3), asList(4, 5, 6))
      .flatMap(numbers -> numbers.stream())
      .collect(toList());
assertEquals(asList(1, 2, 3, 4, 5, 6), together);
```

Streams mapMulti (JDK16+)

- <R> Stream<R> mapMulti(BiConsumer<T, Consumer<R>> mapper)

- Accepts Stream, items of type <T>
- Consumer<R> pushes items to following 'action' in the stream
- Can be used to create multiple items out of 1 stream item.
- Combination of *filter* and *map* with some elements of *flatMap*.

Streams (Aggregations)

	Code
SUM	BigDecimal collect = DB.**root**.**getBooks().stream()** **.collect(** **Collectors.reducing(BigDecimal.ZERO, Book::getPrice, BigDecimal::add));**
GroupBy	Map<Author, BigDecimal> collect = DB.**root**.**getBooks().stream()** **.collect(**Collectors.groupingBy(Book::getAuthor, Collectors.reducing(BigDecimal.**ZERO**, **Book::getPrice, BigDecimal::add)));**

Streams (Sorting)

	Code
Sorting by „String“	DB.root.getBooks().stream() .sorted(Comparator.comparing(Book::getName)) .collect(Collectors.toList());
Sorting by „String“ reversed order	DB.root.getBooks().stream() .sorted(Comparator.comparing(Book::getName).reversed()) .collect(Collectors.toList());
Sorted by „Bigdecimal“	DB.root.getBooks().stream() .sorted((b1,b2) -> b2.getPrice().compareTo(b1.getPrice())) .collect(Collectors.toList());

MicroStream Lazy Loading

Handling large datasets

Topics

- Why do we need to store and load data in a lazy way?
- What data should be stored and loaded lazily?
- How to use lazy references?
- Some best practice examples
- How to release lazy data if no longer needed?

- Sources
  - https://docs.microstream.one/manual/storage/loading-data/lazy-loading/index.html

Why lazy references are important

- MicroStream is an in memory database framework. That's the reason why MicroStream is so fast and performant.
- That means, for best performance the data has to be in the RAM.
- But in some cases it is impossible to hold all the data inside the RAM specially when binary data comes into play.
- And sometimes it is just unnecessary to hold all the data inside the RAM e.g. historical data which is not needed all the time.

Short, we need a way to decide which data should not be loaded to the RAM on MicroStream DB startup!

Why lazy references are important

- By default to whole data will be loaded into the RAM at startup of MicroStream

Why lazy references are important

- With Lazy references it could look like this

When to use lazy loading

- **Always** on binary data
  - Blobs
  - Documents
  - Images
- **Always** on huge collections
  - With less frequented data (e.g. Logs, Outdated data)
- **Sometimes** on collections
  - With a huge amount of sub-entities
- **Whenever you want** if performance isn't that important in this part of data model or the amount of RAM is limited

Usage for binary data

```
public class Book
{
   private String                          isbn;
   private String                          name;
   private Lazy<byte[]>                    image;
   private HashMap<String, Lazy<byte[]>>   documents = new HashMap<>();
   private BigDecimal                      price;
```

- **Single reference to a blob (image)**
  - Use „*image.get()*“ to get the blob out of lazy wrapper
  - Images
- **Collection of images or documents (documents)**
  - Best Practice using a *HashMap* instead of *List / Set* gives possibility to search for the right binary and load it with „*get()*“ in the end.

Huge collections or be aware of huge amount of sub entities

```
public class Store
{
   private String              storeID;
   private Lazy<List<Book>>    books = Lazy.Reference(new ArrayList<Book>());
}
```

- **Easiest way to handle collections with the lazy wrapper**
  - Best practice for most low and medium size collections to be aware of maybe a huge amount of sub items
  - For streaming and filtering you have to load the whole lazy collection at once

Huge collections or be aware of huge amount of sub entities

```
public class Store
{
   private String                                  id;
   private HashMap<String, Lazy<List<Book>>> books = new HashMap<>();
}
```

- **Usage of a HashMap**
  - Best practice for huge collections gives possibility to define an index for books
  - Index could be (A, B, C,...)

Release already loaded data

- There are two strategies to release loaded data. It can be done manually or automatically.
- The “Lazy” class has a “.clear()” method.
- However, such a clear does not mean that the referenced instance immediately disappears from memory. That's the job of the garbage collector of the JVM.
- https://docs.microstream.one/manual/storage/loading-data/lazy-loading/clearing-lazy-references.html#_manually

```
Book book = BookDAO.findBookByISBN("1234567");

// Load image from lazy reference
Lazy<byte[]> lazyImage = book.getImage();
byte[] bs = lazyImage.get();

// release lazy reference
lazyImage.clear();
bs = null;
System.gc(); // JVM Hint, probably ignored, don't write
```

Release already loaded data

- Automatically
  - Each Lazy instance has a “last touched” timestamp. Each “.get()” call set it to the current time. This will tell you how long a Lazy Reference has not been used, i.e. if it is needed at all.
  - The “LazyReferenceManager” audits this. It is enabled by default, with a timeout of 1,000,000 milliseconds, which is about 15 minutes.
  - A custom manager can be set easily, which should happen before a storage is started.

```
LazyReferenceManager.set(LazyReferenceManager.New(
    Lazy.Checker(
        Duration.ofMinutes(30).toMillis(), // timeout of lazy access
        0.75                               // memory quota
    )
));
```


TIME FOR LUNCH

MicroStream
Legacy Type Mapping

What if i have to change my data definition (Classes)?

Topics

- How does MicroStream handle datatypes?
- What happens if we do some changes in the entity?
- What changes can be handled automatically by MicroStream?

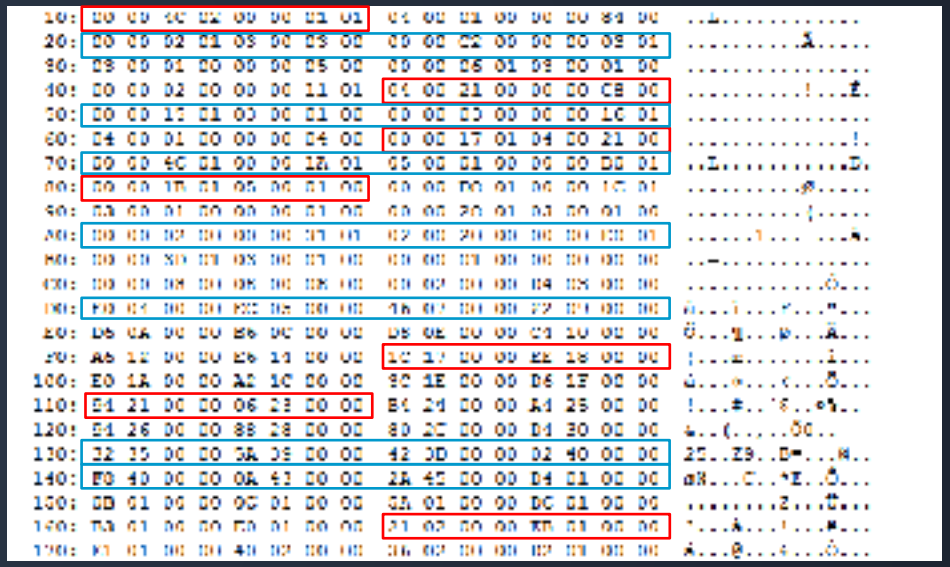
- Sources
  - https://docs.microstream.one/manual/storage/legacy-type-mapping/index.html

How does MicroStream handles Datatypes?

PersistenceTypeDictionary

```
0000000000000000005 int
{
    primitive 32 bit integer signed,
}
...
...
...
0000000000001000066 one.microstream.hackathonuidemo.microstream.domain.Customer
{
    java.lang.String one.microstream.lazydemo.domain.Customer#id,
    java.lang.String one.microstream.lazydemo.domain.Customer#lastname,
    java.lang.String one.microstream.lazydemo.domain.Customer#firstname,
    java.lang.String one.microstream.lazydemo.domain.Customer#mail,
    java.lang.String one.microstream.lazydemo.domain.Customer#gender,
}
```

Storage



- Each entry in the storage has a **TypeID** and a **ObjectID**
- Each type in the storage has its own type entry in the dictionary
- If there is an existing block of binary data in the storage e.g. TypeID 1000066 but this type is missing in the dictionary the storage will not start. **Take care about your “PersistenceTypeDictionary” it has to be in sync.**

What datatypes are supported

- https://docs.microstream.one/manual/storage/addendum/supported-java-features.html
- https://docs.microstream.one/manual/storage/addendum/specialized-type-handlers.html

Lets add an attribute to „Customer“

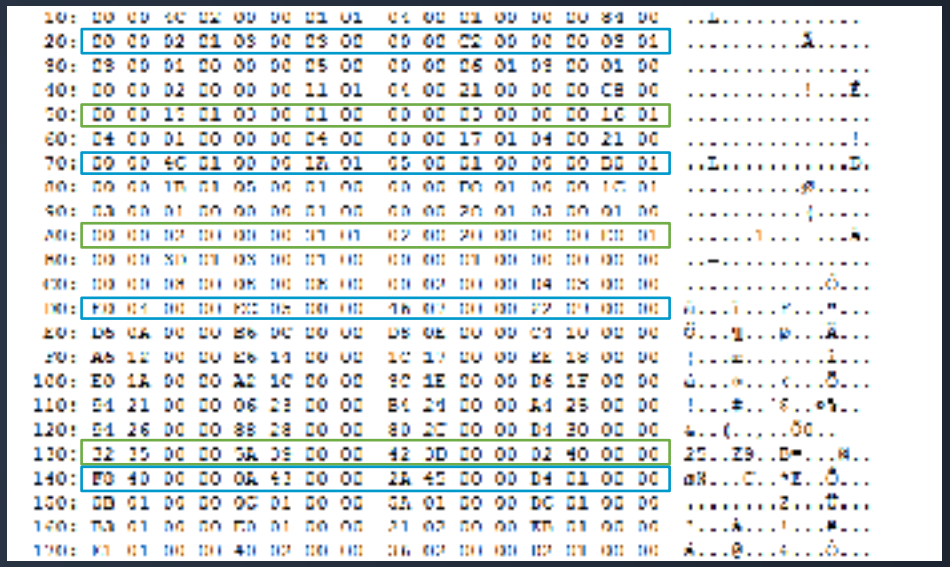
PersistenceTypeDictionary

```
0000000000001000066 one.microstream.hackathonuidemo.microstream.domain.Customer
{
    java.lang.String one.microstream.lazydemo.domain.Customer#id,
    java.lang.String one.microstream.lazydemo.domain.Customer#lastname,
    java.lang.String one.microstream.lazydemo.domain.Customer#firstname,
    java.lang.String one.microstream.lazydemo.domain.Customer#mail,
    java.lang.String one.microstream.lazydemo.domain.Customer#gender,
}
0000000000001000067 one.microstream.hackathonuidemo.microstream.domain.Customer
{
    java.lang.String one.microstream.lazydemo.domain.Customer#id,
    java.lang.String one.microstream.lazydemo.domain.Customer#lastname,
    java.lang.String one.microstream.lazydemo.domain.Customer#firstname,
    java.lang.String one.microstream.lazydemo.domain.Customer#mail,
    java.lang.String one.microstream.lazydemo.domain.Customer#gender,
    java.lang.String one.microstream.lazydemo.domain.Customer#ipAddress,
}
```

„OLD“

„NEW“

Storage



- After adding, removing or moving attributes a new Type with a TypeID occurs in the dictionary
- Every-time an „OLD“ object is loaded it will be automated transformed to the „NEW“ version of the object
- **Keep in mind, after removing an attribute the data is also gone!**

What changes can be handled automatically?

- Basically the heuristic attempts to automatically detect which fields are new, have been removed, reordered or altered.

Contact.java (old)

```
1  public class Contact
2  {
3      String name     ;
4      String firstname;
5      int    age      ;
6      String email    ;
7      String note     ;
8      Object link     ;
9  }
```

Contact.java (new)

```
1  public class Contact
2  {
3      String         firstname    ; // moved
4      String         lastname     ; // renamed
5      String         emailAddress ; // renamed
6      String         supportNode  ; // renamed
7      PostalAddress  postalAddress; // new
8      int            age          ; // moved
9  }
```

What changes can be handled automatically?

```
Console Output

1  ----------
   Legacy type mapping required for legacy type
2  10000055:Contact
   to current type
   1000056:Contact
   Fields:
   java.lang.String Contact#firstname  1.000     > java.lang.String Contact#firstname
3  java.lang.String Contact#name       -0.750 ----> java.lang.String Contact#lastname
4  java.lang.String Contact#email      -0.708 ----> java.lang.String Contact#emailAddress
   java.lang.String Contact#note       -0.636 ----> java.lang.String Contact#supportNote
5  [***new***] PostalAddress Contact#postalAddress
6  int Contact#age                     -1.000 ----> int Contact#age
7  java.lang.Object Contact#link [discarded]

   Write 'y' to accept the mapping.
```

What changes can be handled automatically?

What the heuristic is doing now is something like this:

- String "**firstname"** is equal in both classes, so it has to be the same, pretty much as int age.
- "**name**" and "**lastname**" is pretty similar, type is the same too. If there is nothing better for the two, they probably belong together. Same with the other two fields.
- In the end, the ominous "**link"** and "**postalAddress"** remain. The heuristic can not make sense of that, so it assumes that one thing falls away and the other one is added.

In this particular example, that worked perfectly. Well done, heuristic.

Best Practice

- During the development when we are able to delete the storage and let it recreate this should be first choice. We would recommend to create some mockup classes and data to create a nice and clear storage on startup.
- Changes of multiple attributes in one class should be done step by step instead of doing them all at once.
- Creating a backup of the storage before updating should be natural.
- Keep in mind, datatype changes requires a special migration logic in most cases and can't be done automatically. In this particular case:
  1. First create the new attribute
  2. Start the application and transfer the data from the old attribute to the new attribute.
  3. Shutdown application and remove old attribute
  4. Done

Example

- https://github.com/rdebusscher/microstream-legacy-type-mapping

Demo

MicroStream

MicroStream Backup strategies

What if storage fails?

Backup storage

- To protect against corruption
- Only serialised once
  - Backup storage has same config
    - Channel count
    - Naming
    - “Blob” sizes
    - …
- Doesn’t need to be same type of AFS
  - Mixture of Disk and Database possible

Backup Strategies

- Full Backup
  - Issued on request
  - Copies storage target content
  - Impacts application throughput
  - StorageManager.issueFullBackup();

- Incremental backup
  - Performed continues
  - After writing binary data to main storage
    - Write to backup storage

Demo

Crash safe

- StorageManager is crash safe
  - When method returns, all data is persisted
  - *StorageManager::shutdown* is not required

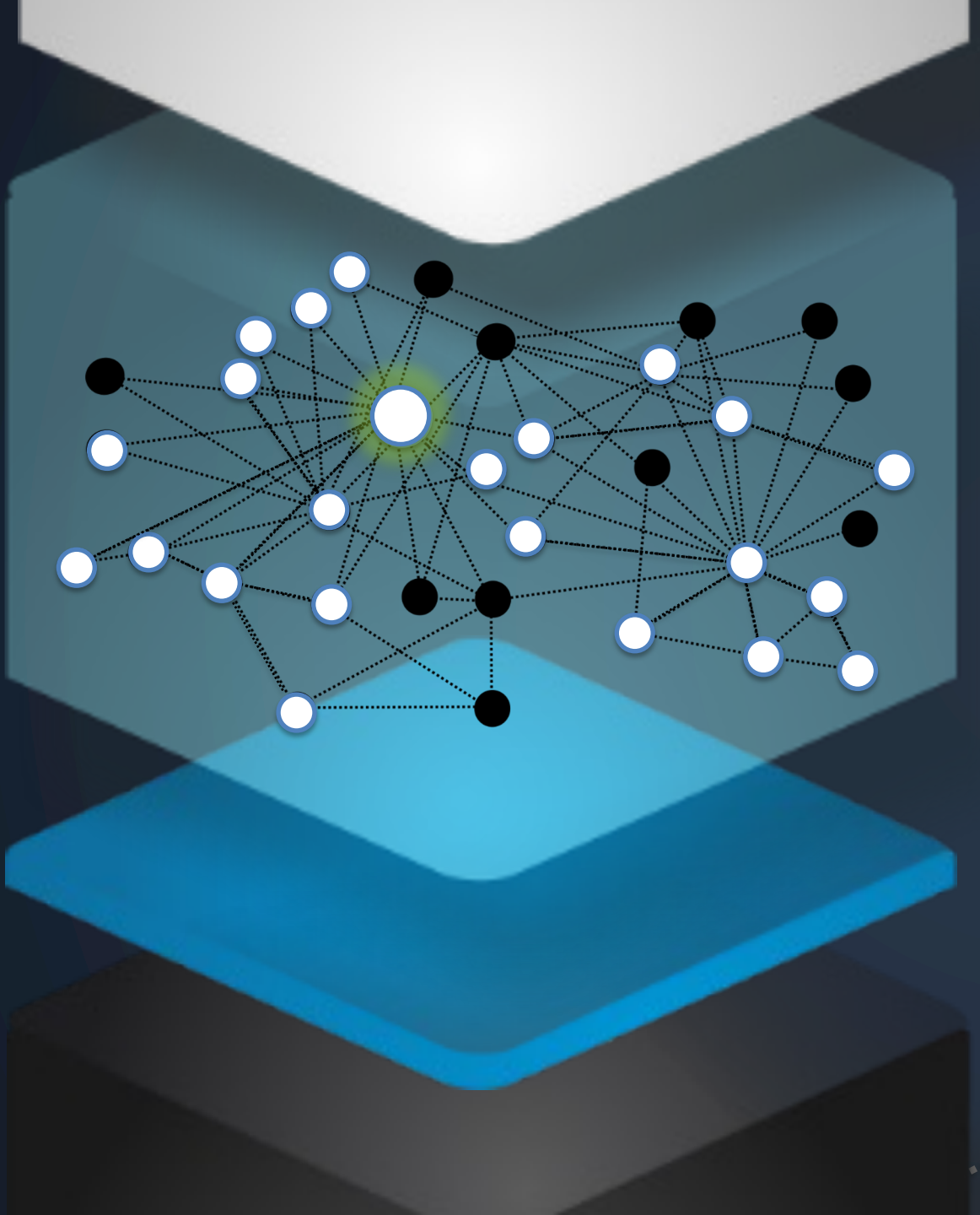
- Not all API calls are crash safe
  - Eager/Lazy-Storer::store -> commit() required to save the data.

MicroStream Housekeeping process

Appending changes

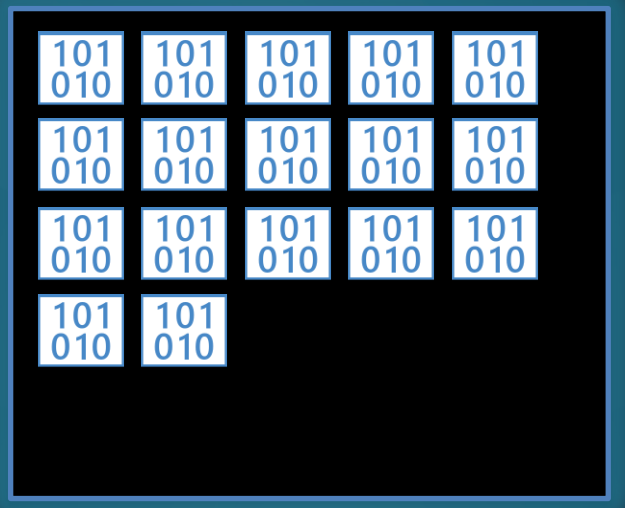
- Original block marked as obsolete
- Appended to the end of storage
  - Performance, no file rearrangement required

- Additional action(s) required to keep storage ‘as small as possible’



App / MicroStream

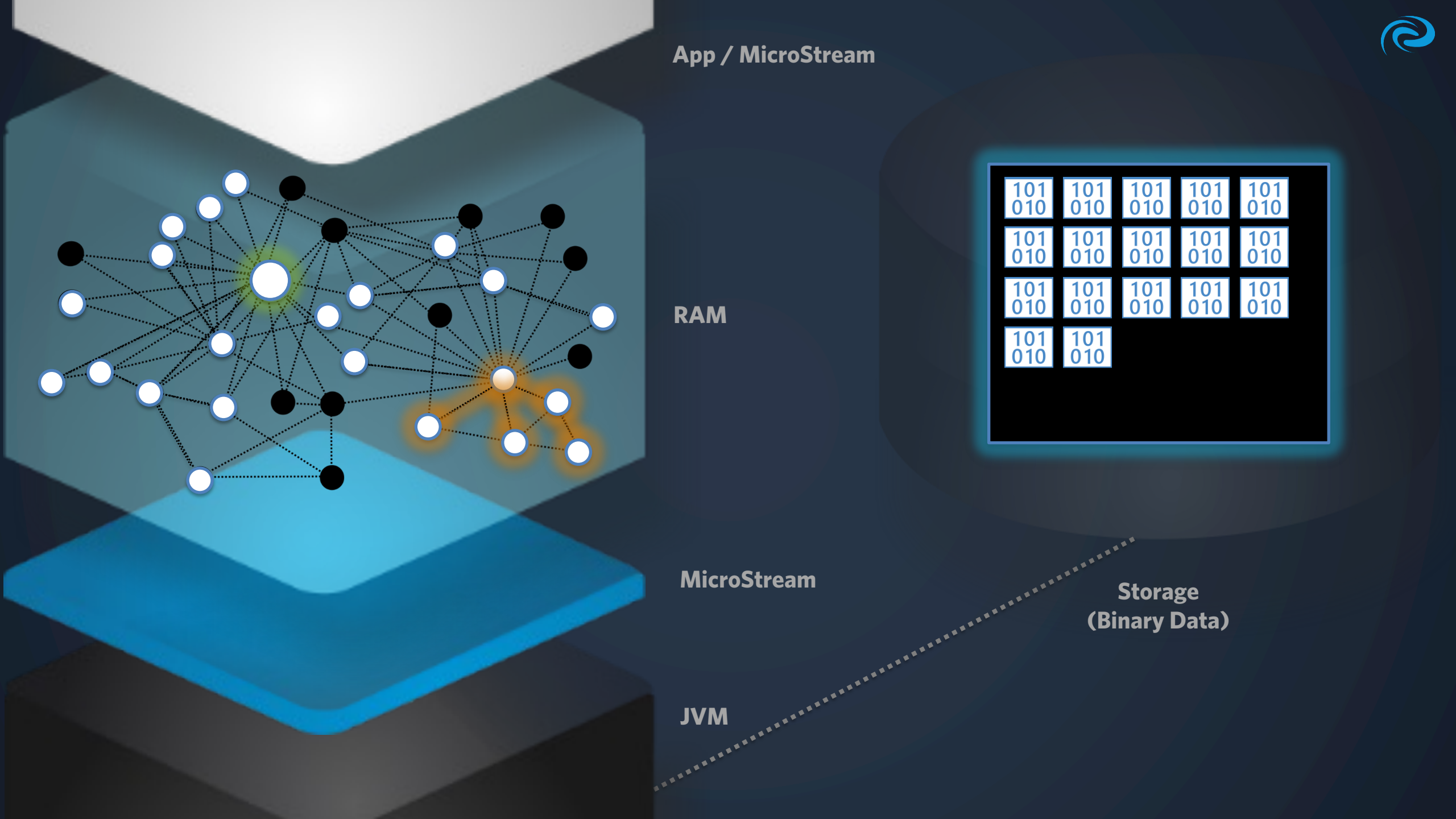
RAM

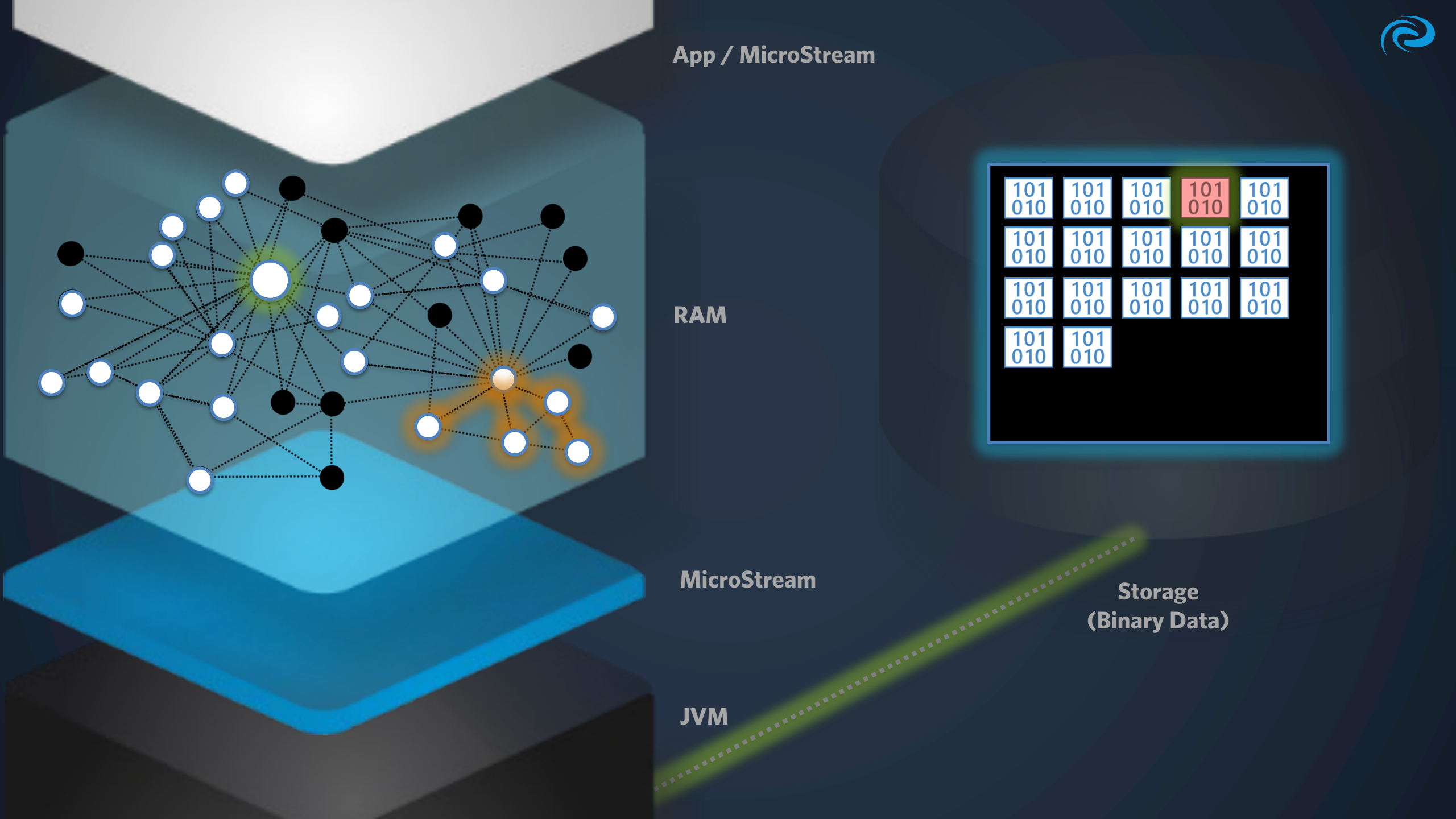


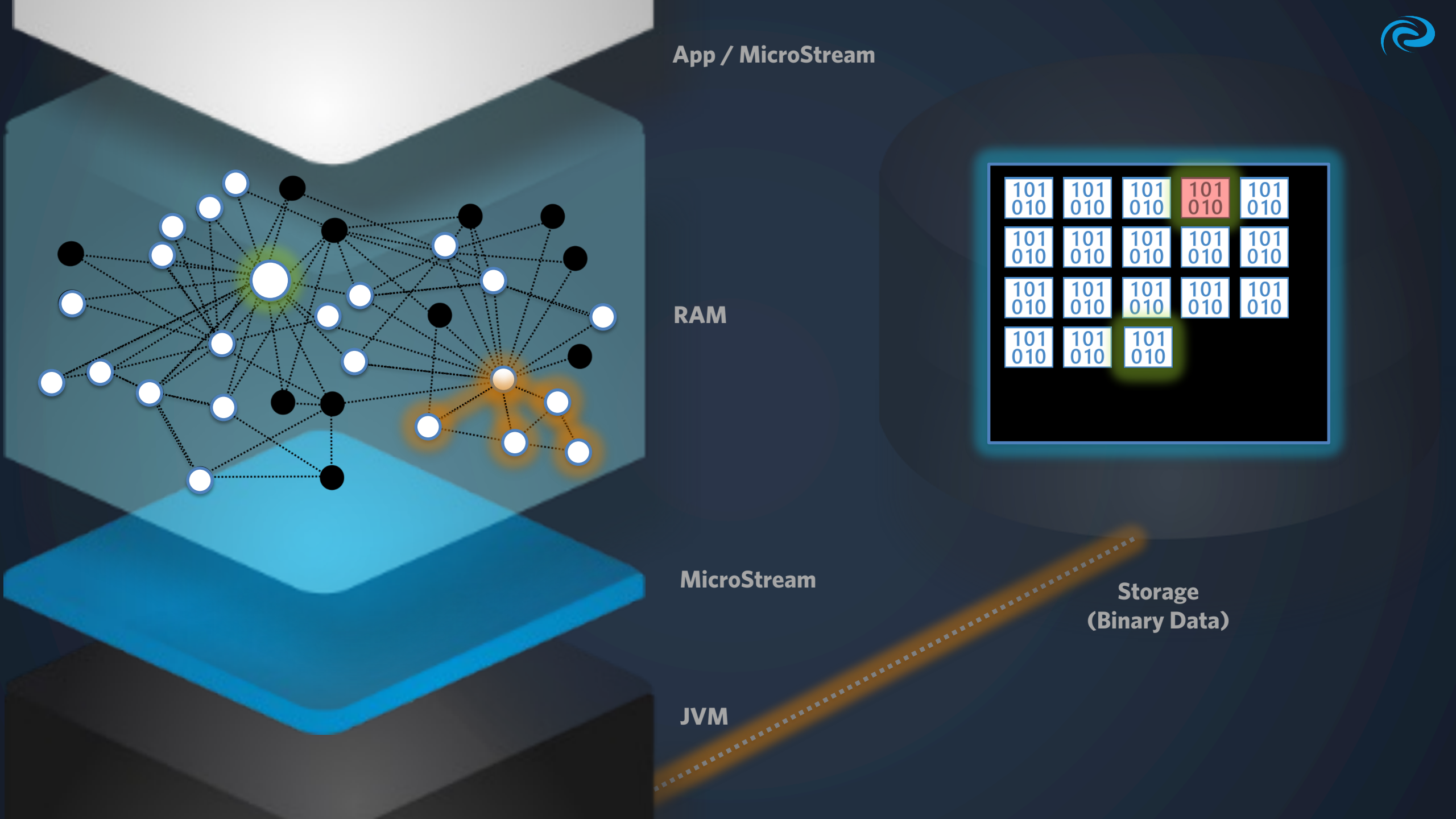
MicroStream

Storage
(Binary Data)

JVM


App / MicroStream
RAM
101 010
MicroStream
Storage
(Binary Data)
JVM


App / MicroStream
RAM
101 010
MicroStream
Storage
(Binary Data)
JVM


App / MicroStream
RAM
101 010
MicroStream
Storage
(Binary Data)
JVM

Housekeeping (1)

- Several distinct processes

- Within Channel
  - storage Cleanup
  - StorageHousekeepingController, interval and time budget

- Within Lazy Reference Manager
  - Prepare Lazy references for JVM GC.
  - Last access and Memory pressure based
  - Lazy.Checker(
    Duration.ofMinutes(30).toMillis(), // timeout of lazy access
    0.75 // memory quota
)

Housekeeping (2)

- Garbage Collection
  - Cleanup unused references
  - Similar to the JVM one
  - Within storage ‘files’ and internal structures (like cache)

MicroStream

MicroStream Integrations

2 types of integrations

Inclusion

Helidon (2.4 +)
Micronaut (3.5 +)

Integration

Spring Boot
Jakarta / MicroProfile Config
Quarkus

Integration

- Configure using configuration values
  - Using framework specific config
    - Spring Boot Config
    - MicroProfile Config
    - SmallRye Config (part of Quarkus)

- Customise through 'beans'
  - Interfaces *EmbeddedStorageFoundationCustomizer* and *StorageManagerInitializer*

- Inject Root and StorageManager

Inclusions

- Helidon
  - https://medium.com/helidon/microstream-helidon-brave-performance-combination-12d3d288aa
- Micronaut
  - https://micronaut-projects.github.io/micronaut-microstream/snapshot/guide/

MicroStream

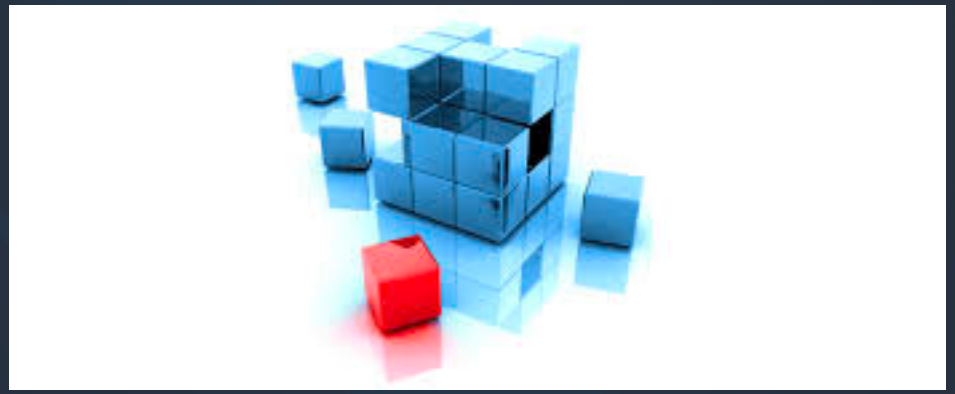
MicroStream Best Practices

Development

- Review data model for optimal access to collections
  - Include Lazy where needed
  - ‘Indexing’ where needed (Map structures)
- Proper concurrent access strategy
- Exception and rollback strategy
- No need to use Legacy Type Mapping feature
- Don’t activate backup
- Initial data set through mockup (load test data)

Production

- Update channel count if needed
- Tune housekeeping processes
  - Change budget time
- Activate backup strategy
  - Useful for migration data model tests
- Properly test data model migration
  - Using Legacy Type Mapping and production data backup



Advanced Training

- https://javapro.io/training/

Date	Duration	Time	Region	Course	
September 27 - 28, 2022	2 Days	09:00 - 15:00 CEST	Europe, Middle East, Africa	MicroStream Fundamentals	Details
October 13, 2022	1 Day	09:00 - 15:00 CEST	Europe, Middle East, Africa	MicroStream Advanced	Details
October 27, 2022	1 Day	10:00 - 17:00 CDT	North America and South America	MicroStream Advanced	Details
November 8 - 9, 2022	2 Days	09:00 - 15:00 CET	Europe, Middle East, Africa	MicroStream Fundamentals	Details
November 15 - 16, 2022	2 Days	10:00 - 17:00 CST	North America and South America	MicroStream Fundamentals	Details
December 8, 2022	1 Day	09:00 - 15:00 CET	Europe, Middle East, Africa	MicroStream Advanced	Details
December 15, 2022	1 Day	10:00 - 17:00 CST	North America and South America	MicroStream Advanced	Details

- Booking code : MICROSTREAM-2022

Advanced Training agenda

- Creating an advanced configuration of the MicroStream storage
- Deeper knowledge about advanced topics like “Housekeeping” or “Storage Channels”
- Creating proper data structures for eager and lazy data handling
- Usage or transactions and rollbacks with MicroStream
- Exception handling with MicroStream
- Extended usage of LazyLoading with Apache Lucene
- Concurrency handling within a MicroStream application
- MicroStream customizations
  - Custom type handler
  - Custom class loader
  - Custom LegacyTypeHandler
  - Custom store behaviour
- Different store target like AWS S3 or MySQL