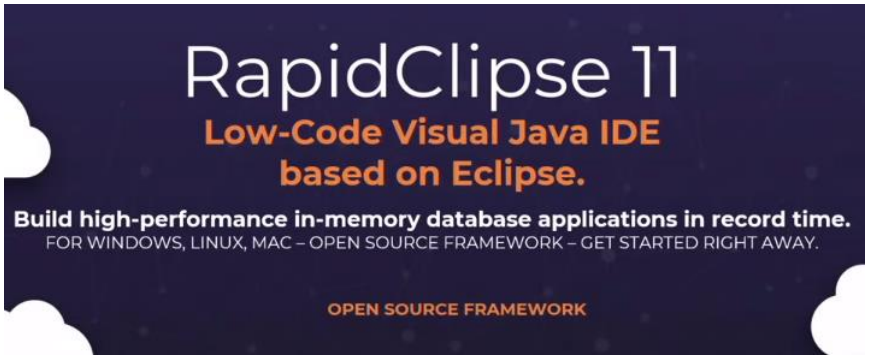
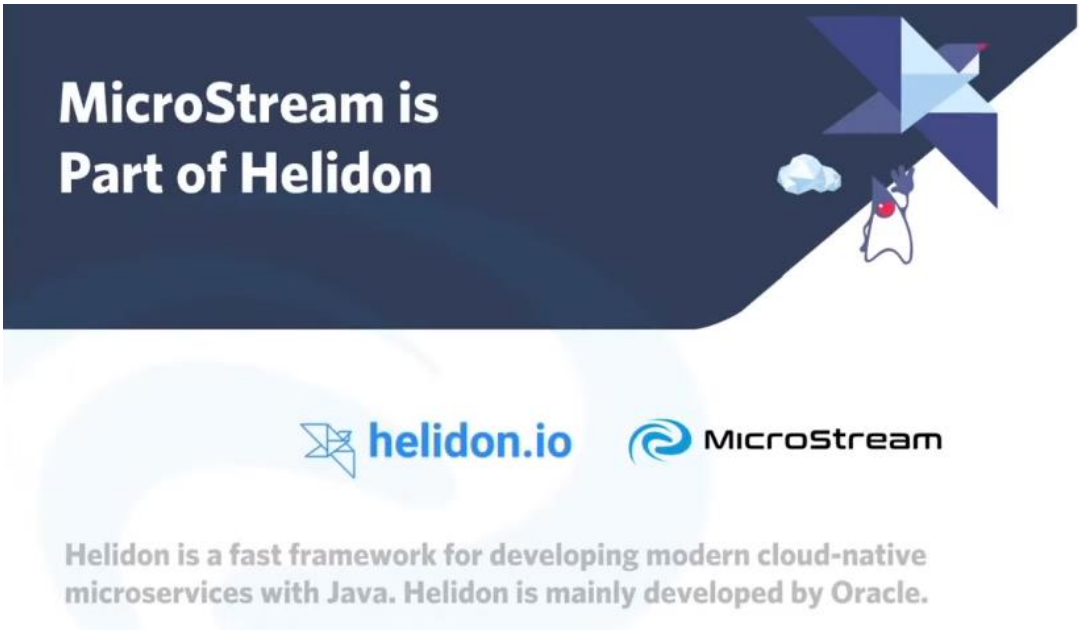
Java has been leading for more than 20 years. However, Java is known as complicated, heavy-weight, slow, and for its slow startup time and high RAM consumption. Now, the cloud age has begun and the cloud changes everything. Is Java still suited for the cloud? Are traditional databases still suited for the cloud or is NoSQL the way to go? Can in-memory computing speed up my system and how does Java support it? Is serverless really interesting for Java developers and how do Java and serverless fit together?

No worries, the holy grail was already found. A completely new Java stack was created to build real cloud-native apps with Java. Native images with Java, millisecond startup-time, in-memory data processing, up to 1000x faster database queries, highly cost-efficient serverless infrastructure, everything built with pure Java. That's not a vision, it's reality. In this session, you will learn how to build ultra-fast cloud-native apps and microservices with core Java and modern Java micro-frameworks only.



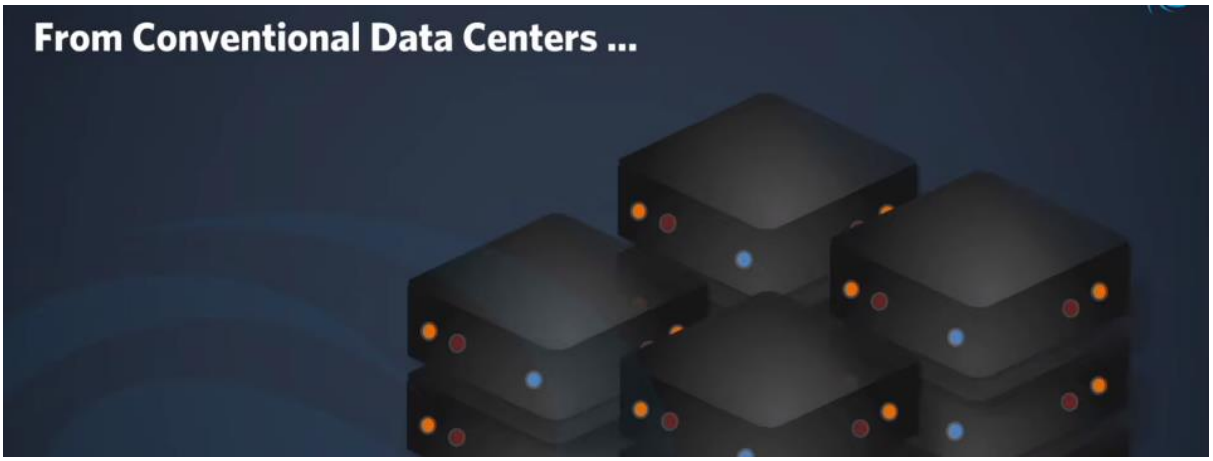


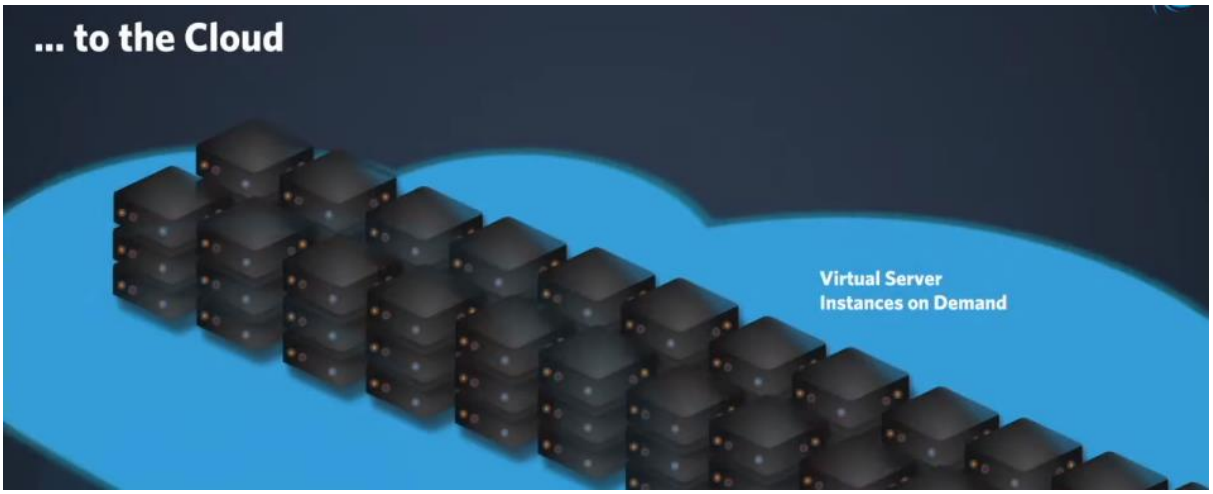
MICRONAUT™

MicroStream is Integrated with Micronaut

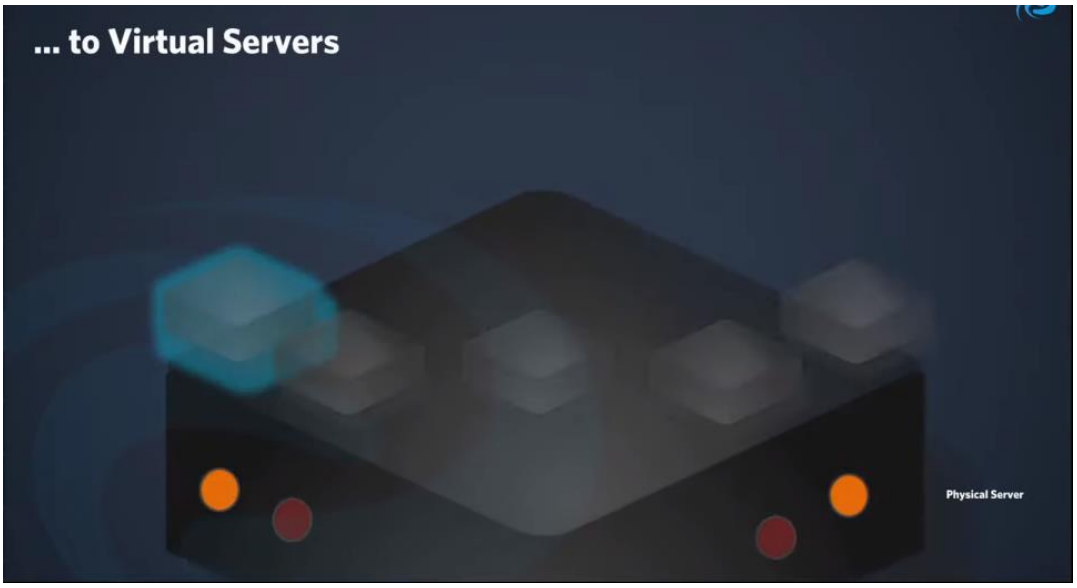
Micronaut is a modern, JVM-based, full-stack framework for building modular, easily testable microservice and serverless applications.

Cloud has changed everything

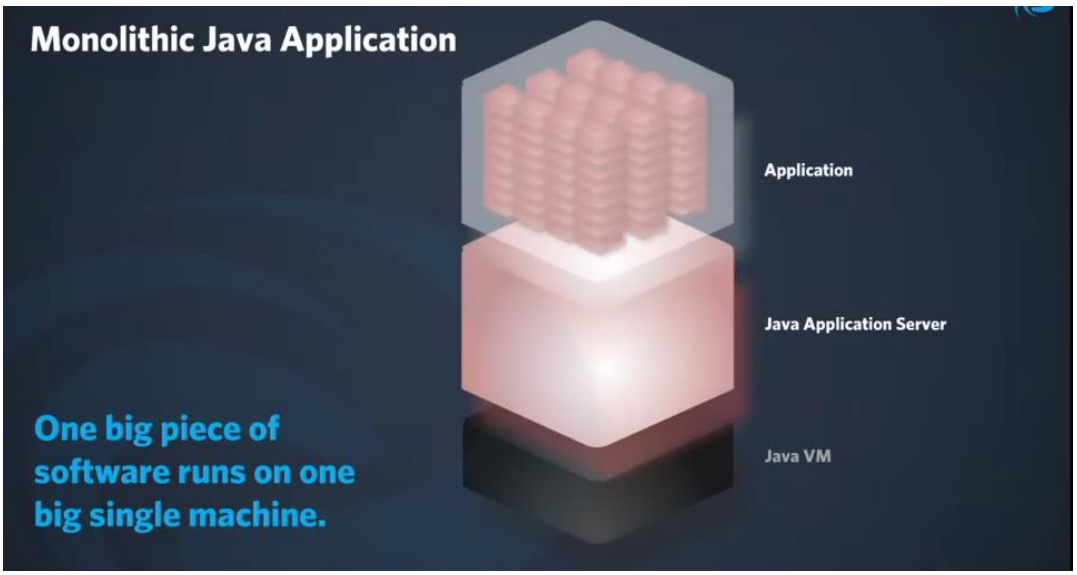


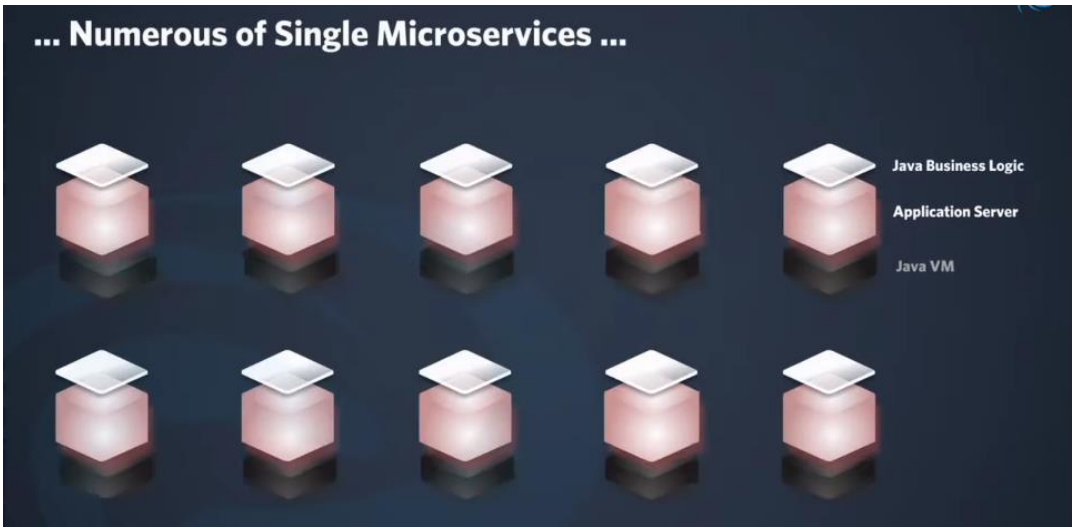


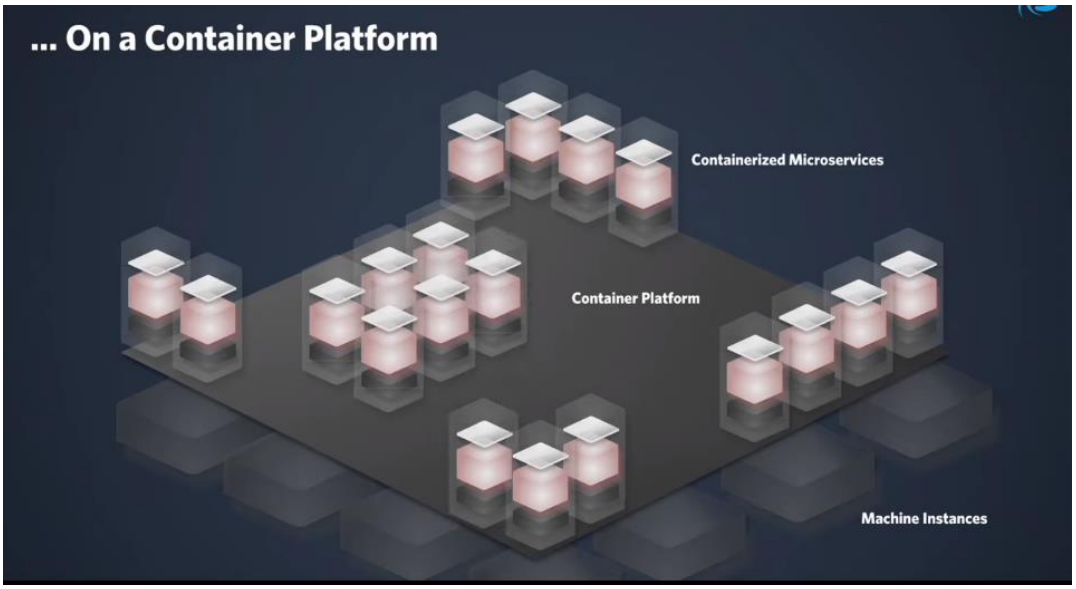
Virtualization



MicroServices



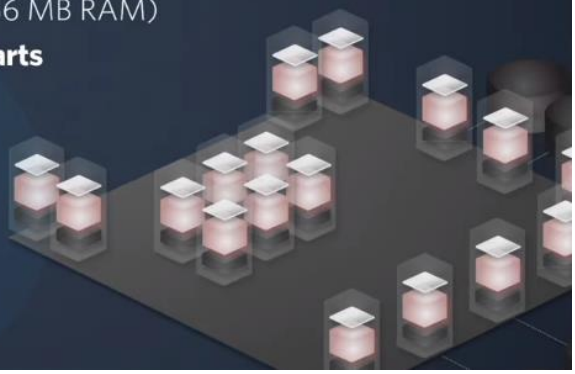




Challenges with Microservices Using the Traditional Java Stack

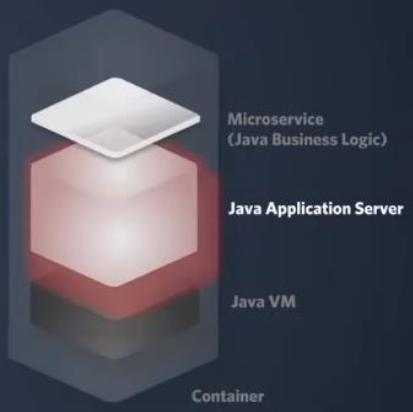
Environment & Requirements to Microservices

- **Responsible for only 1 specific task** (separation of concerns)
- **Beeing as small as possible** (micro application)
- **Containerized - running in its own container**
- **Running on micro machines** (0.1 CPU, 256 MB RAM)
- **Must be built for regular crashes & restarts**
- **Fast startup time**
- **Low memory consumption**
- **Has its own database**



Challenges with Traditional Java Application Server

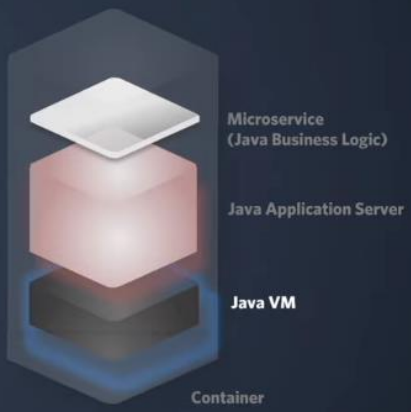
- **Built for running big monolithic Java applications**
- **Running multiple Java applications**
- **Sharing resources**
- **Slow startup time**
- **High memory consumption**
- **Heavyweight**



Should each microservice has its own application server?

Challenges with Traditional Java VM

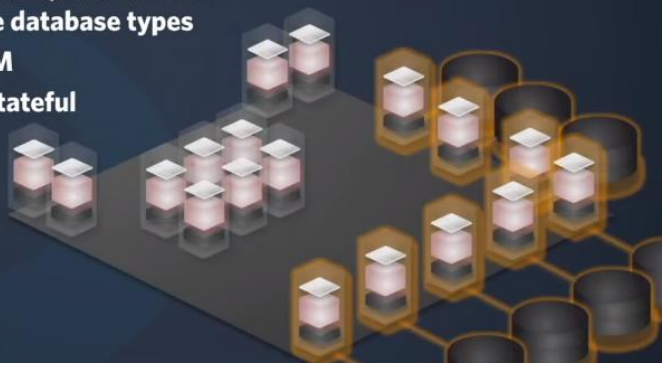
- **Built to run for long time periods**
- **Becomes extremely fast, but only after JIT optimization intervals** (Warm-up)
- **Slow startup time**
- **High memory consumption**



Startup time is too slow for microservices.

Challenges with Traditional Java Persistence

- Each microservice that stores data has ist own database
- The database should fit best to the specific domain model of the microservice. Thus, microservice infrastructure need multiple database types
- Each microservice uses ORM
- Database connections are stateful
- High devops effort
- DBaaS is expensive

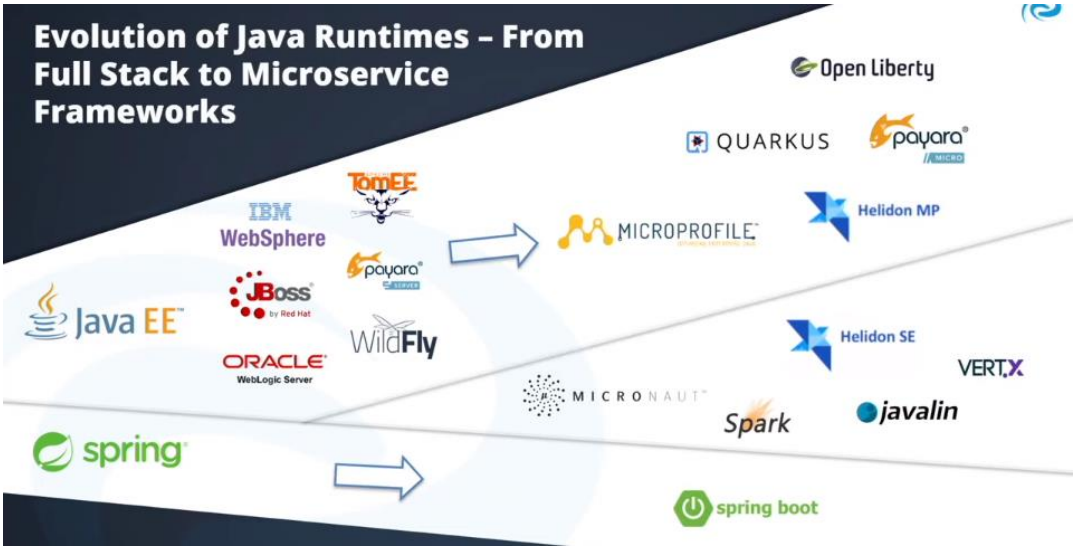


From Java Application Servers to Microservice Frameworks / Micro Runtimes

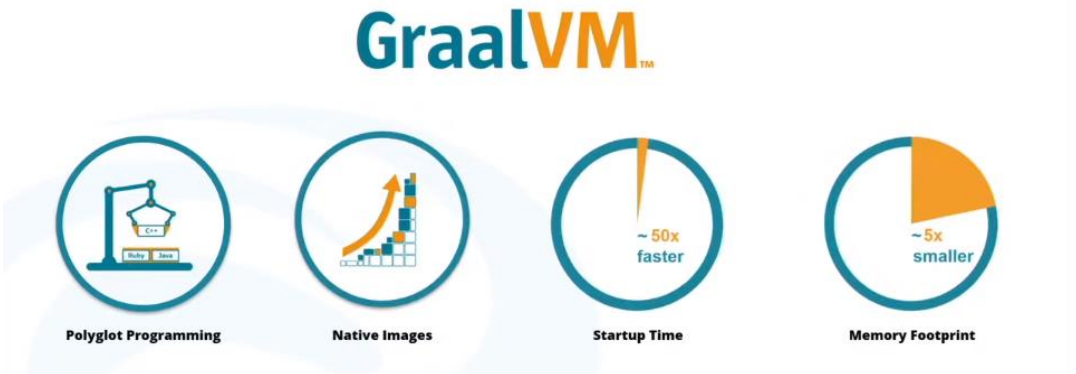
From Application Servers ..

to Micro Runtimes (Microservice Framework)

From Application Servers to Micro Runtimes



From JVM to Native Images



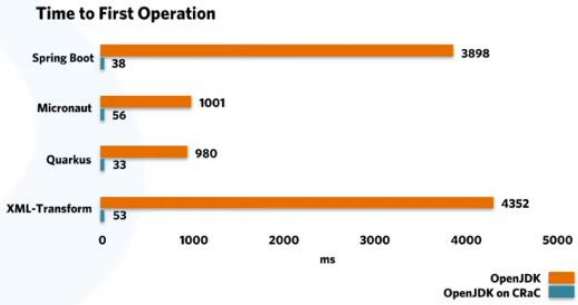
Limitations to GraalVM Native Execution

- No JVMTI, Java Agents, JMX, JFR support
- Efficient only for smaller heap
- Reflection usage must be known at build time
- Ahead-of-time compiling is linked to longer build times
- Target platform must be known at build time
- Performance - JIT is faster than a native image
- No thread & heap dump support

OpenJDK CRaC

The CRaC (Coordinated Restore at Checkpoint) Project researches coordination of Java programs with mechanisms to checkpoint (make an image of, snapshot) a Java instance while it is executing. Restoring from the image could be a solution to some of the problems with the start-up and warm-up times. The primary aim of the Project is to develop a new standard mechanism-agnostic API to notify Java programs about the checkpoint and restore events. Other research activities will include, but will not be limited to, integration with existing checkpoint/restore mechanisms and development of new ones, changes to JVM and JDK to make images smaller and ensure they are correct.

Azul ReadyNow
OpenJ9 InstantOn
AWS Corretto SnapStart

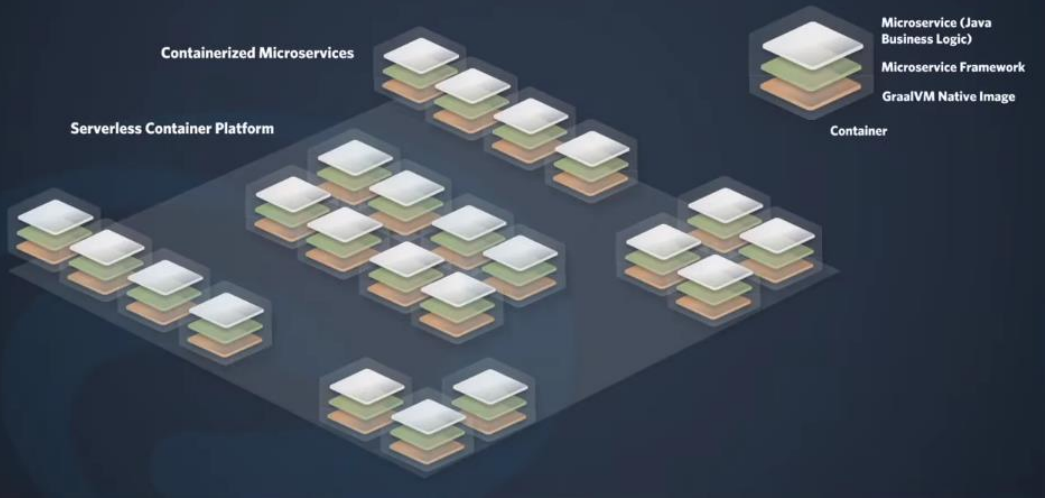


This creates and stores a snapshot of your warmed up JVM (native or non-native) to allow faster startup times later

Serverless Infrastructure ...

Serverless Computing

Modern Java Microservice Serverless Architecture



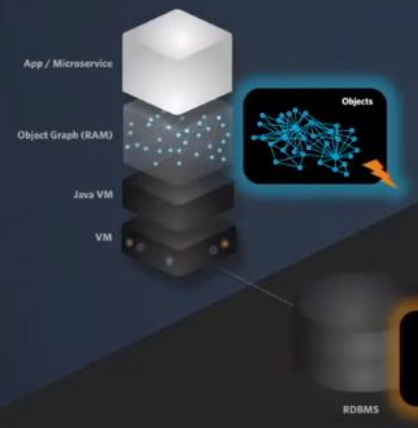
Serverless Functions (e.g. AWS Lambda)

- Serverless computing platform
- Event-driven architecture
- Fully automated management of computer resources required by code
- No infrastructure configuration
- Supported programming languages: NodeJS, Python, Go, Ruby, C#, Java
- No DevOps effort
- Linux container, 128 MB - 10 GB RAM, 512 MB - 10 GB storage
- Pay only for what you use, price is based on RAM size (GB) and execution time in ms
- Idle resources are not charged
- 1M requests per month, 400,000 GB-seconds of compute time per month

Java Persistence ...

From Traditional Java Persistence to Micro Persistence

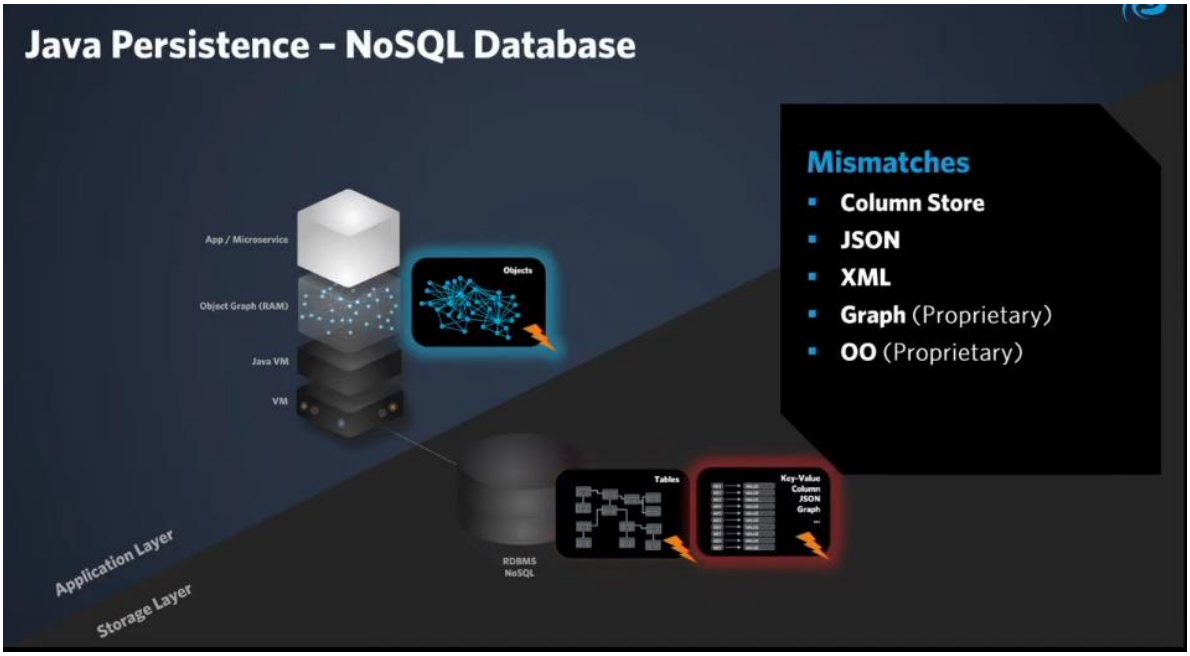
Java Persistence - Object-Relational Impedance Mismatch



Mismatches

- Granularity mismatch
- Subtypes mismatch
- Identity mismatach
- Associations mismatch
- Data Navigation mismatch
- Data type differences

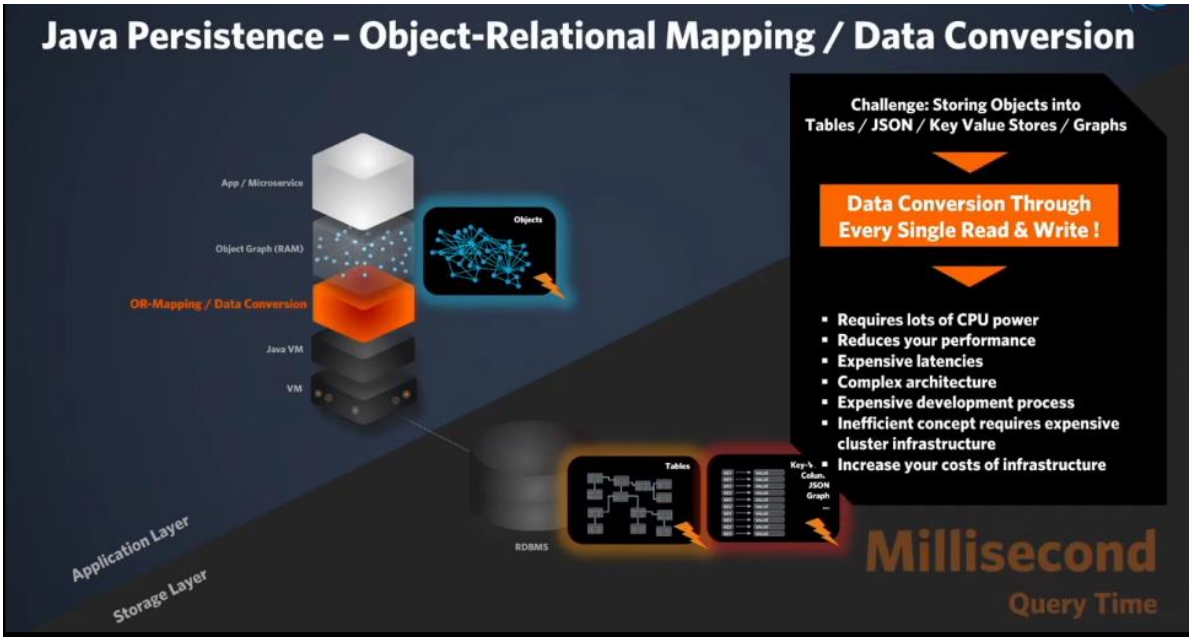
Java Persistence – NoSQL Database



Mismatches

- **Column Store**
- **JSON**
- **XML**
- **Graph** (Proprietary)
- **OO** (Proprietary)

Java Persistence – Object-Relational Mapping / Data Conversion



Challenge: Storing Objects into Tables / JSON / Key Value Stores / Graphs

Data Conversion Through Every Single Read & Write !

- **Requires lots of CPU power**
- **Reduces your performance**
- **Expensive latencies**
- **Complex architecture**
- **Expensive development process**
- **Inefficient concept requires expensive cluster infrastructure**
- **Increase your costs of infrastructure**

Millisecond
Query Time

The Problem of Incompatible Data Structures is Well Known as Impedance Mismatch

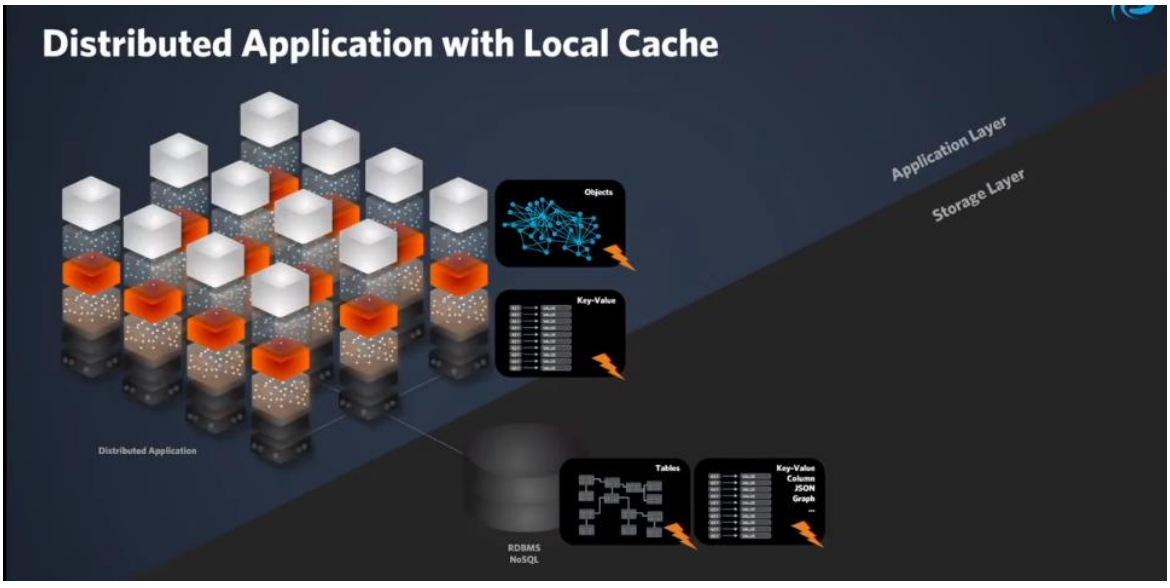
WIKIPEDIA
The Free Encyclopedia

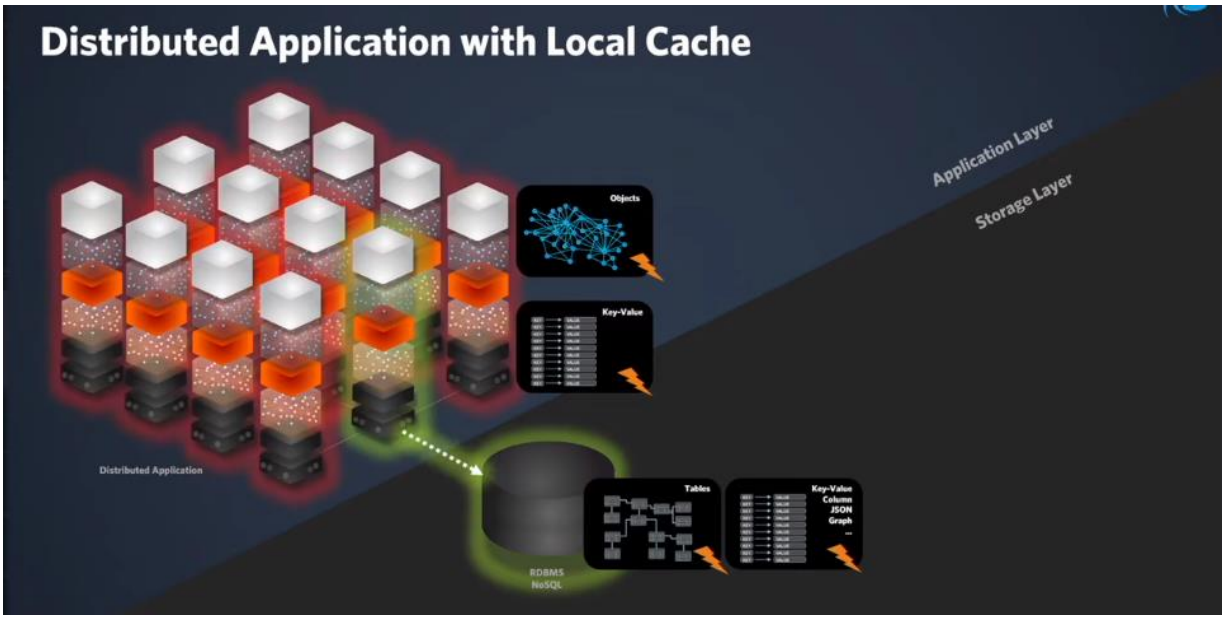
There are various solutions, but they are only a more or less elegant way around the problem. No matter which solution you choose - as long as the systems are different, every developer will sooner or later get to the point where his solution no longer meets one or more of the following points: Maintainability, performance, intelligibility.

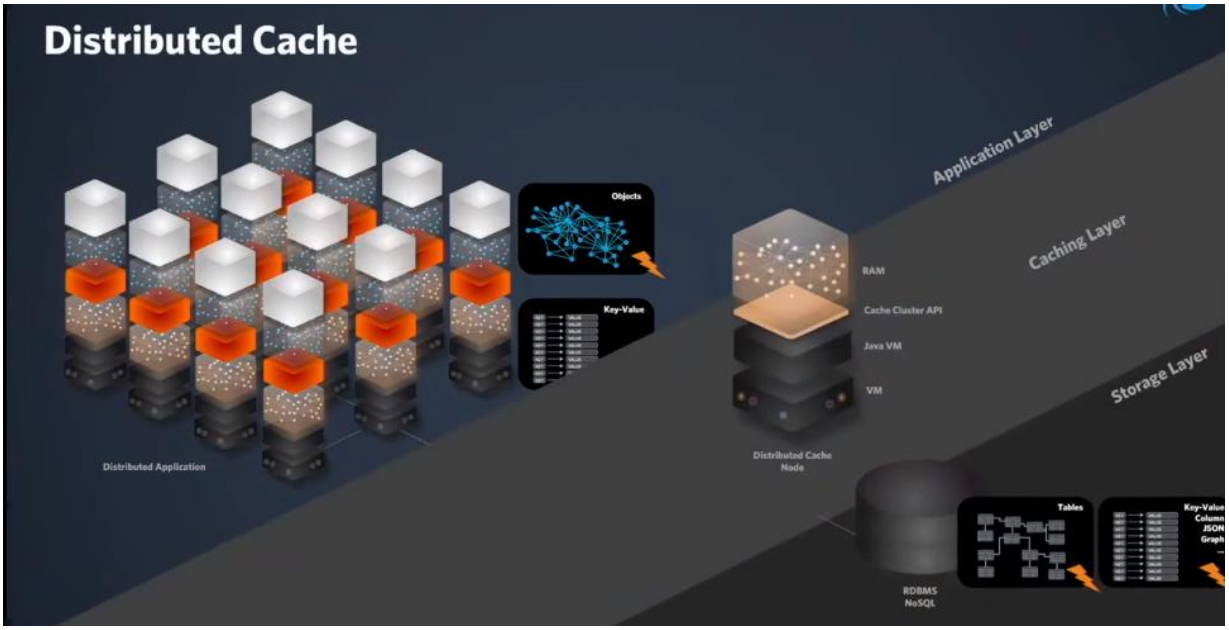
https://en.wikipedia.org/wiki/Object%E2%80%93relational_impedance_mismatch

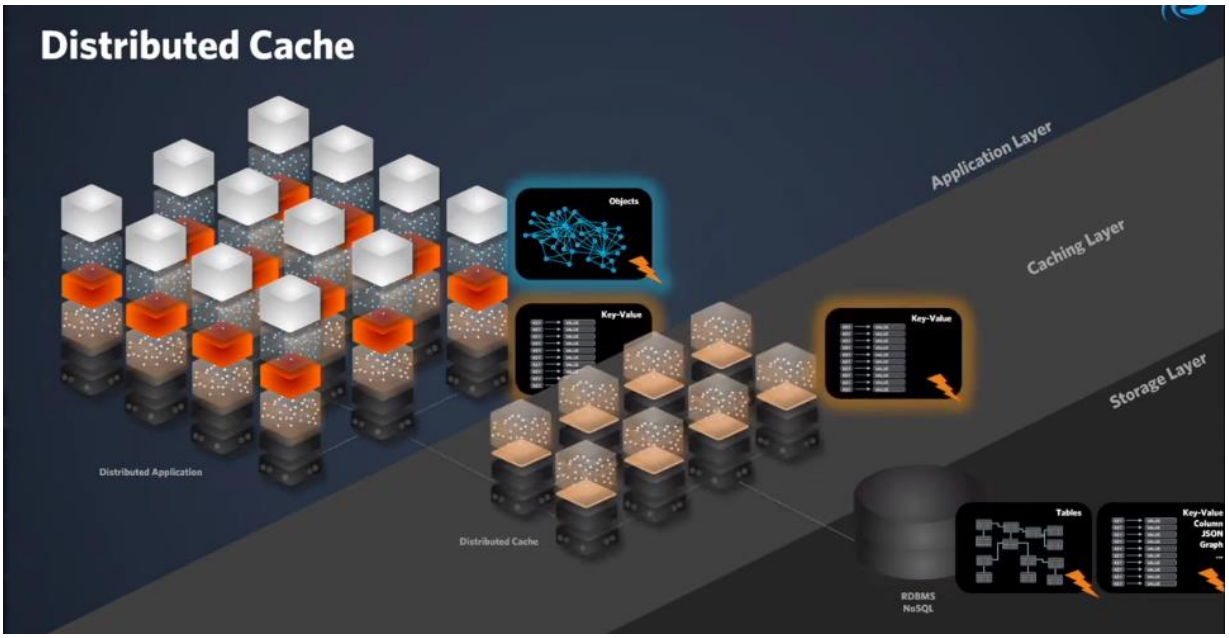

Java Persistence
App / Microservice
Object Graph (RAM)
OR-Mapping / Data Conversion
Java VM
VM
Objects
Tables
Key-Value
Column
JSON
Graph
...
RDBMS
NoSQL
Application Layer
Storage Layer

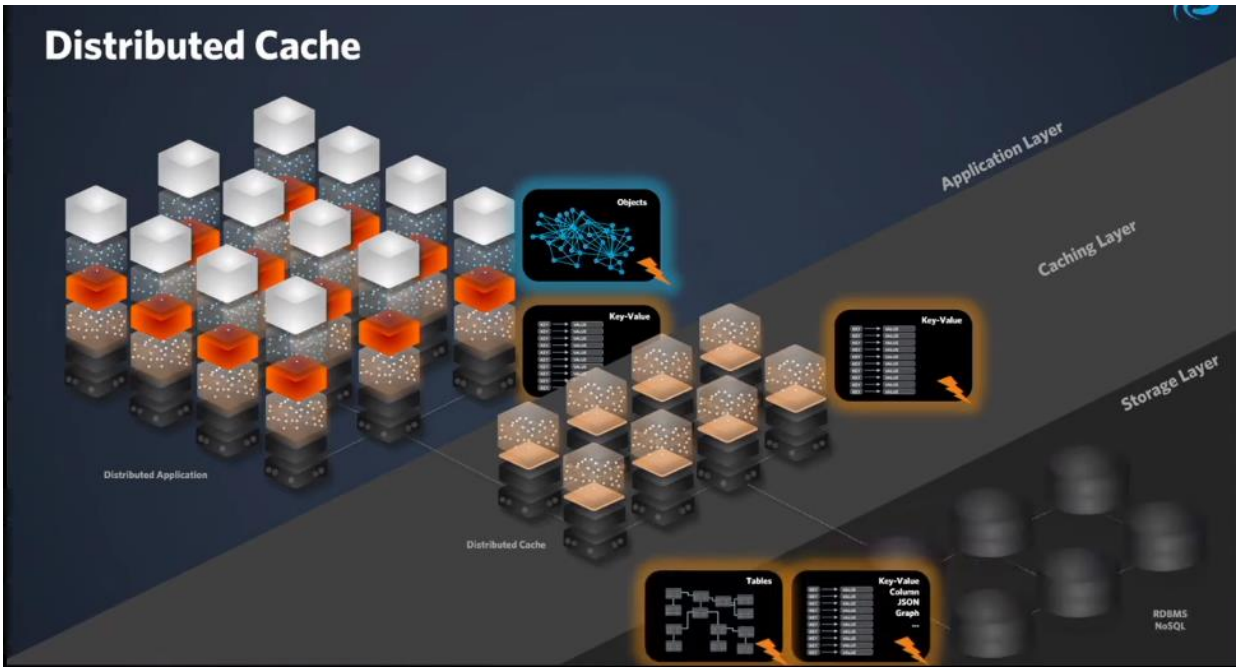

Local Cache
App / Microservice
Object Graph (RAM)
OR-Mapping / Data Conversion
Local Cache
Java VM
VM
Objects
Key-Value
Tables
Key-Value
Column
JSON
Graph
...
RDBMS
NoSQL
Application Layer
Storage Layer


Distributed Application with Local Cache
Distributed Application
Objects
Key-Value
Tables
Key-Value
Column
JSON
Graph
...
RDBMS
NoSQL
Application Layer
Storage Layer


Distributed Application with Local Cache
Application Layer
Storage Layer
Objects
Key-Value
Distributed Application
Tables
Key-Value
Column
JSON
Graph
...
RDBMS
NoSQL


Distributed Cache
Application Layer
Caching Layer
Storage Layer
Objects
Key-Value
Distributed Application
RAM
Cache Cluster API
Java VM
VM
Distributed Cache
Node
Tables
Key-Value
Column
JSON
Graph
...
RDBMS
NoSQL


Distributed Cache
Application Layer
Caching Layer
Storage Layer
Objects
Key-Value
Distributed Application
Key-Value
Distributed Cache
Tables
Key-Value
Column
JSON
Graph
...
RDBMS
NoSQL

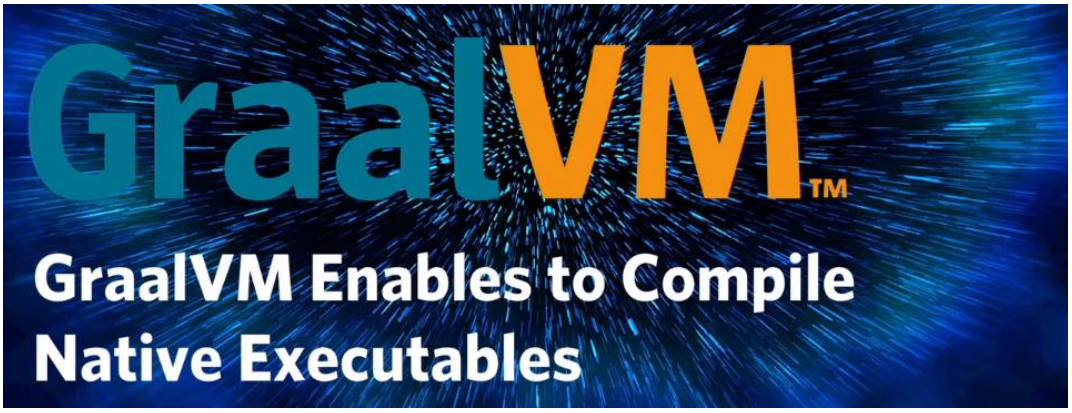


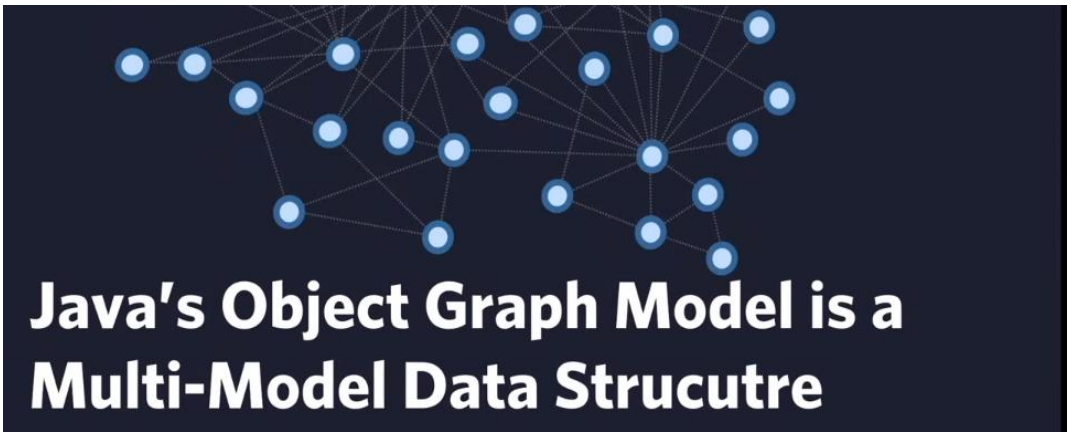
Java In-Memory Data Processing

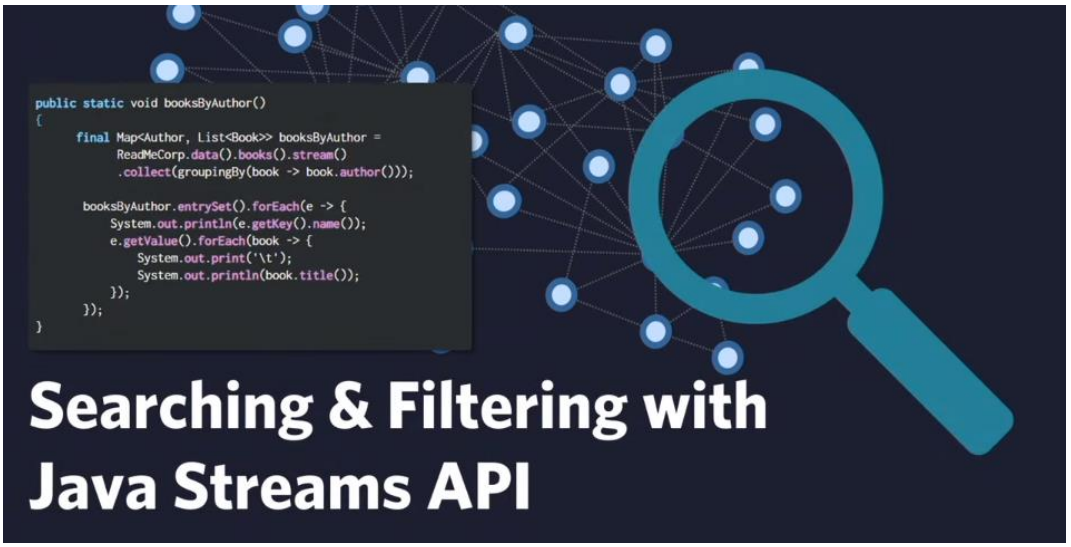
Latency Numbers Every Programmer Should Know

```
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
```

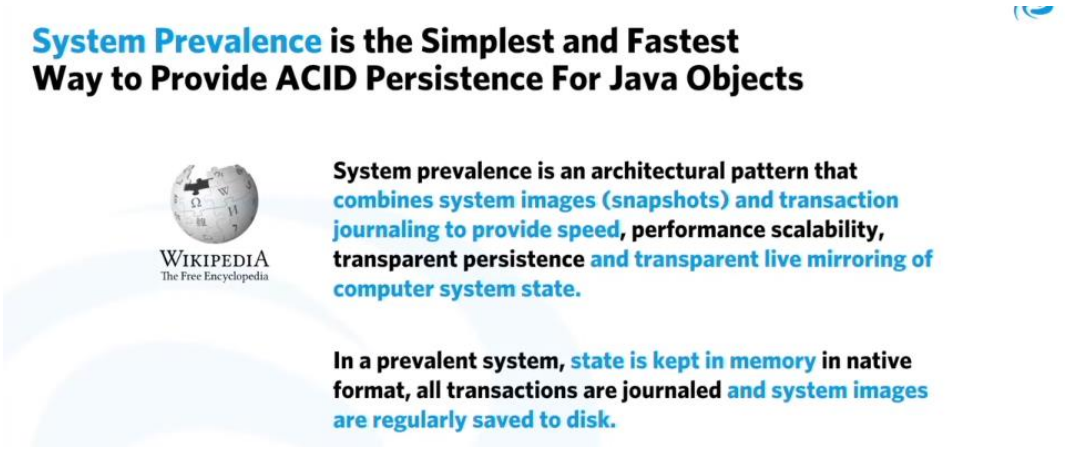
Source: https://gist.github.com/jboner/2841832

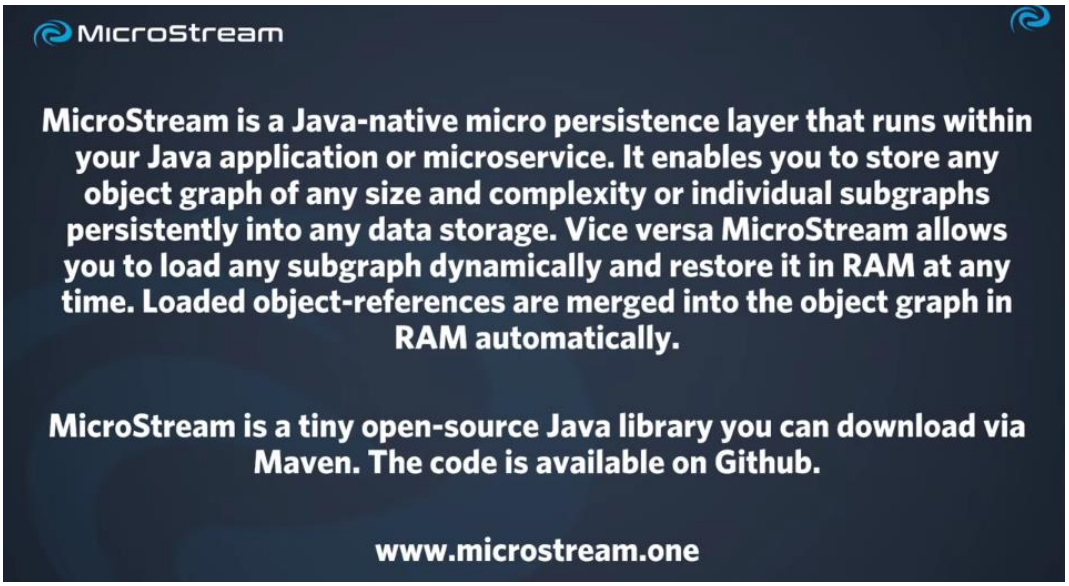

GraalVM™
GraalVM Enables to Compile
Native Executables


Java's Object Graph Model is a
Multi-Model Data Strucutre

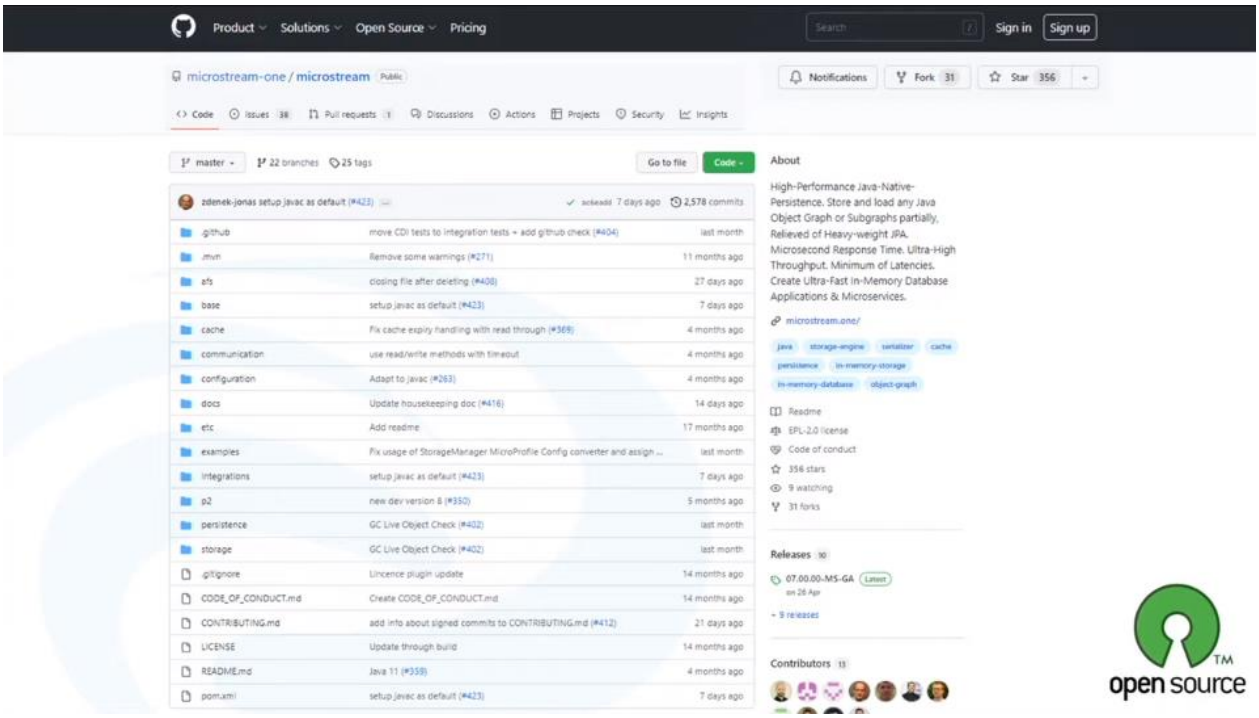

public static void booksByAuthor()
{
final Map<Author, List<Book>> booksByAuthor =
ReadMeCorp.data().books().stream()
.collect(groupingBy(book -> book.author()));
booksByAuthor.entrySet().forEach(e -> {
System.out.println(e.getKey().name());
e.getValue().forEach(book -> {
System.out.print('\t');
System.out.println(book.title());
});
});
}
Searching & Filtering with
Java Streams API


1000x
All Databases: Millisecond Query Time
Java Streams API: Microsecond Query Time





It just serializes your Java data object graphs in-memory and into a blob store like S3 which you can search with the Java Streams API and also restore it easily when needed.



Maven Download

```xml
<repositories>
    <repository>
        <id>microstream-releases</id>
        <url>https://repo.microstream.one/repository/maven-public/</url>
    </repository>
</repositories>
<dependencies>
    <dependency>
        <groupId>one.microstream</groupId>
        <artifactId>storage.embedded</artifactId>
        <version>04.01.00-MS-GA</version>
    </dependency>
    <dependency>
        <groupId>one.microstream</groupId>
        <artifactId>storage.embedded.configuration</artifactId>
        <version>04.01.00-MS-GA</version>
    </dependency>
</dependencies>
```

MicroStream is an Integral Part of Helidon & Micronaut

MicroStream is proud to be an integral part of and contribute to leading microservice frameworks. We are happy to deliver value to the Java Open Source community.

MICRONAUT®

First-class support for MicroStream has been added to the Micronaut framework, and MicroStream is now a major sponsor of the Micronaut Open Source project.

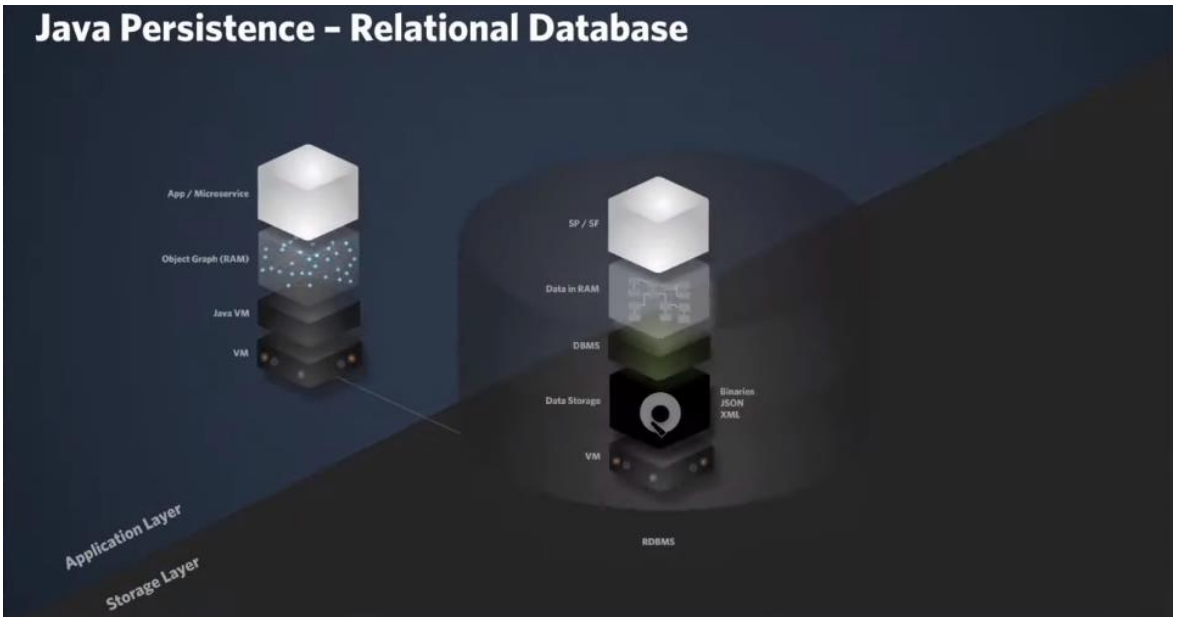
helidon.io

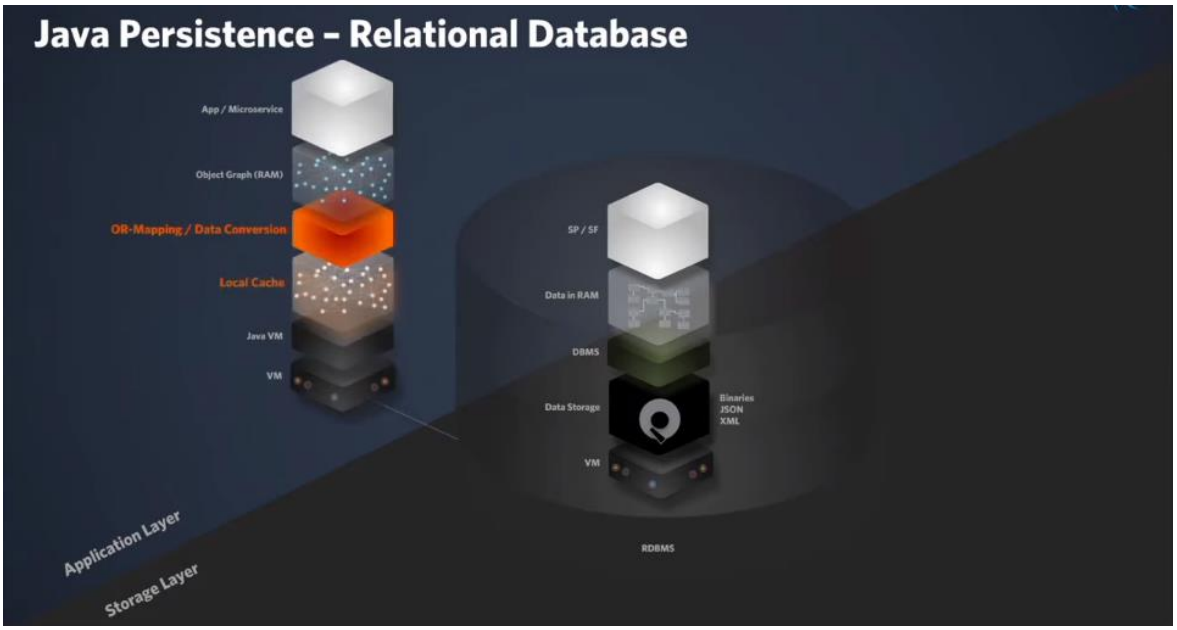
A seamless MicroStream integration has been added to both Helidon flavours, Helidon MP and SE. Helidon is a leading microservice framework for Java, powered by Oracle.

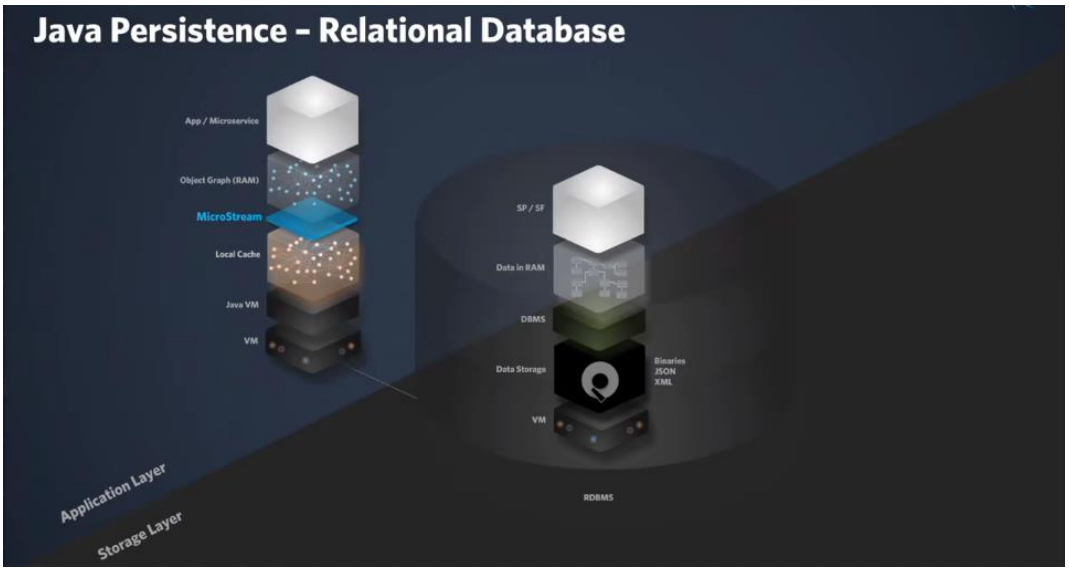
Our Partners

azul ORACLE IBM payara

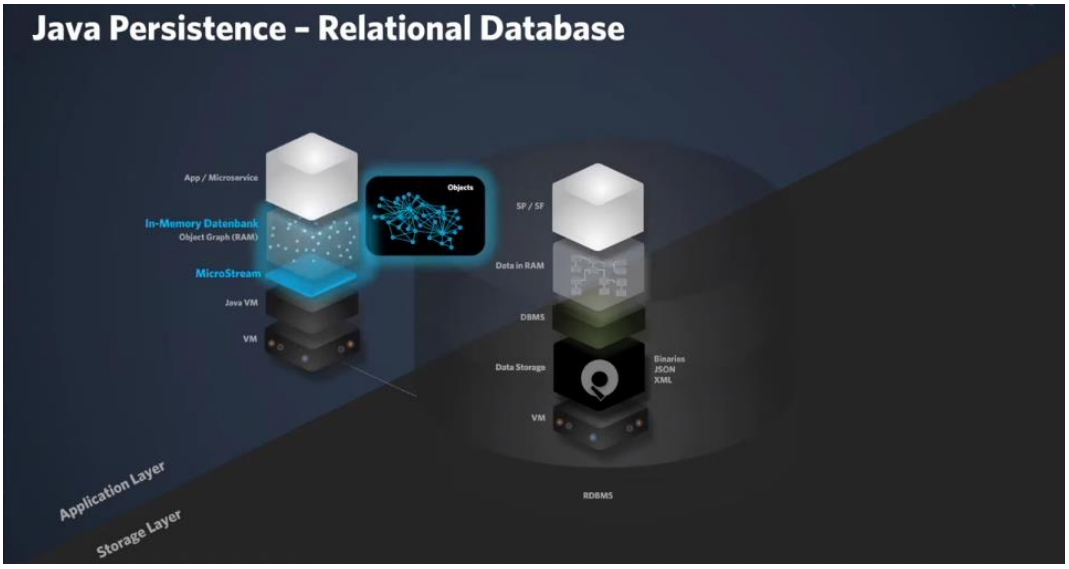

Java Persistence – Relational Database
App / Microservice
Object Graph (RAM)
Java VM
VM
RDBMS
Application Layer
Storage Layer


Java Persistence – Relational Database
App / Microservice
Object Graph (RAM)
Java VM
VM
SP / SF
Data in RAM
DBMS
Data Storage
Binaries
JSON
XML
VM
RDBMS
Application Layer
Storage Layer

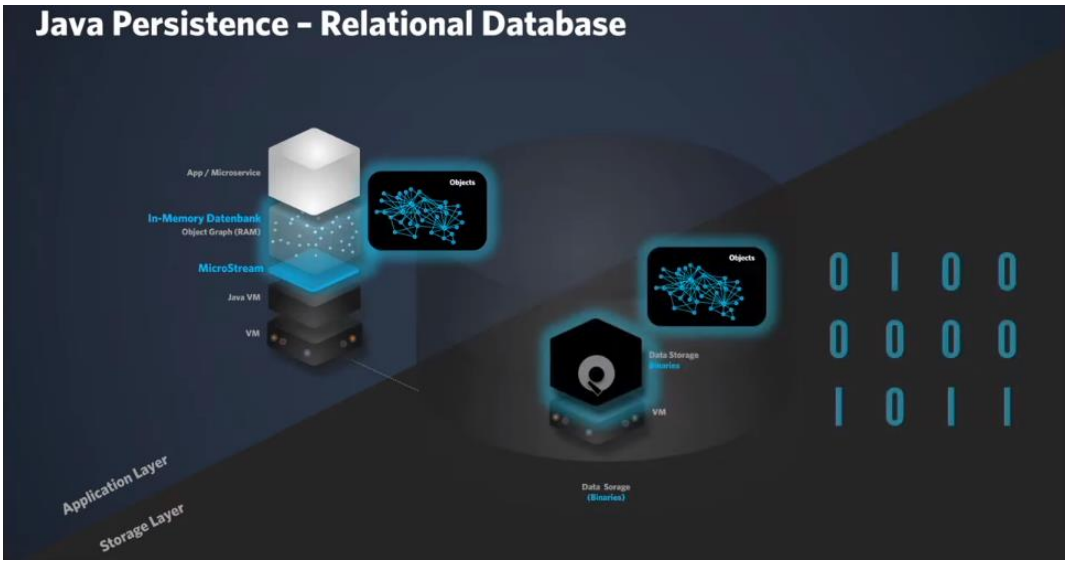

Java Persistence – Relational Database
App / Microservice
Object Graph (RAM)
OR-Mapping / Data Conversion
Local Cache
Java VM
VM
SP / SF
Data in RAM
DBMS
Data Storage
Binaries
JSON
XML
VM
RDBMS
Application Layer
Storage Layer



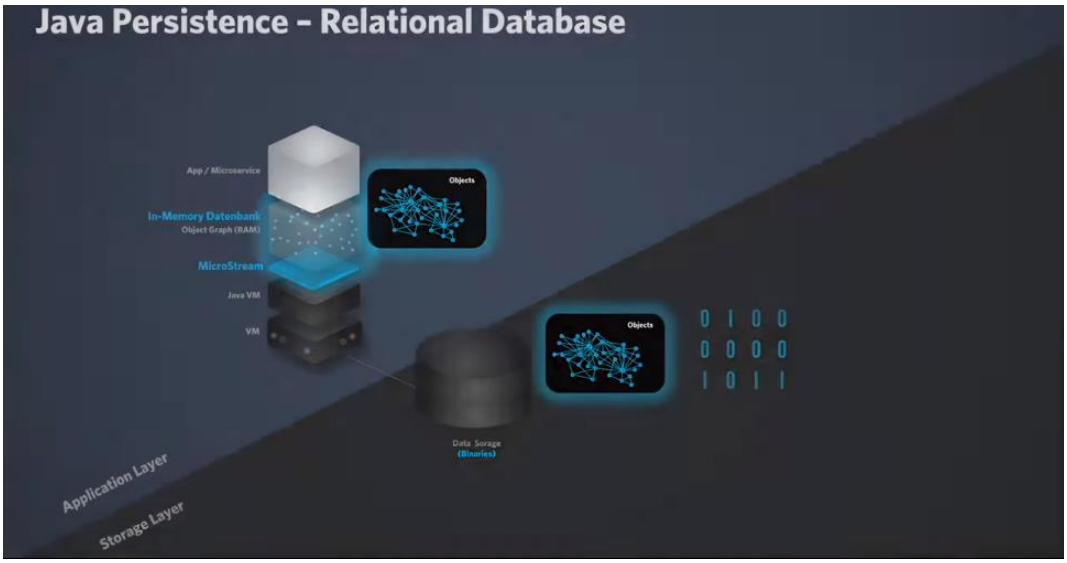
We have replaced object relational mapping with just serialization with **MicroStream**

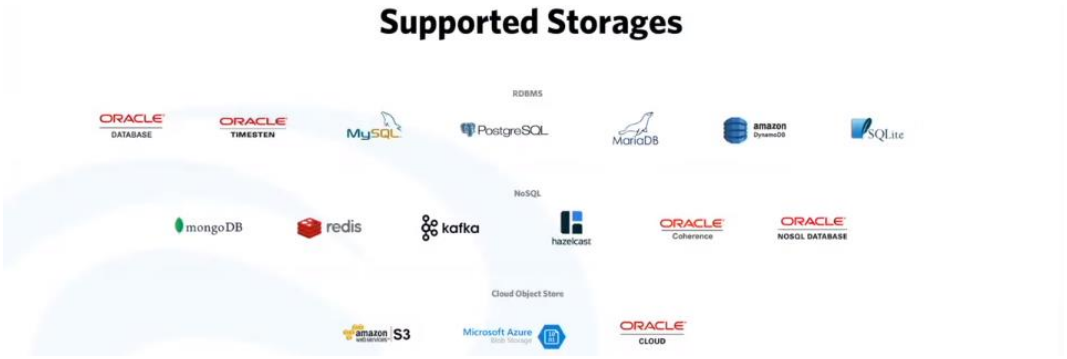


We then keep the data in-memory as a Java native in-memory database (this eliminates the need for a data cache!).

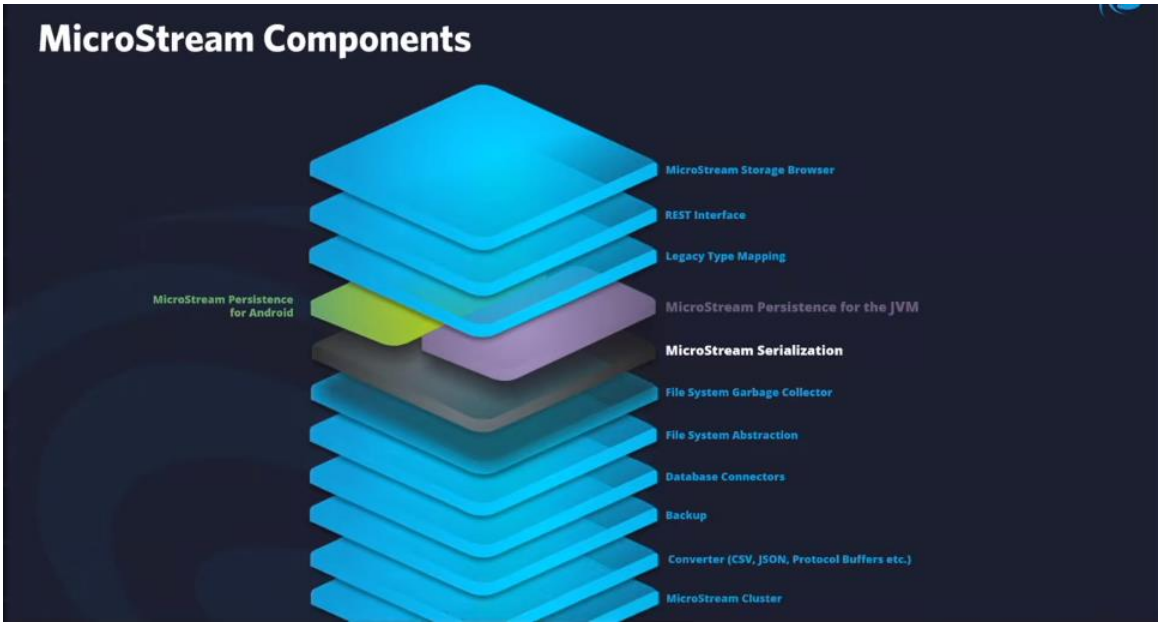


We now do not need data on the persistence side anymore but just replace it with the storage.

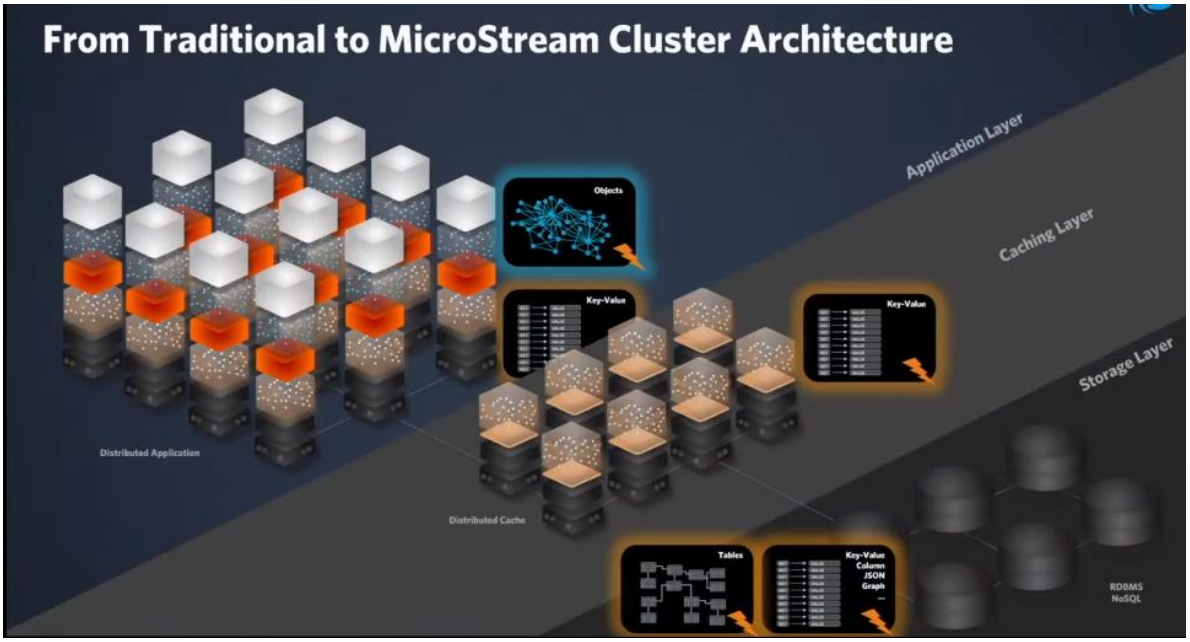

Java Persistence – Relational Database
App / Microservice
Objects
In-Memory Datenbank
Object Graph (RAM)
MicroStream
Java VM
VM
Objects
0 1 0 0
0 0 0 0
1 0 1 1
Data Storage
(Binaries)
Application Layer
Storage Layer

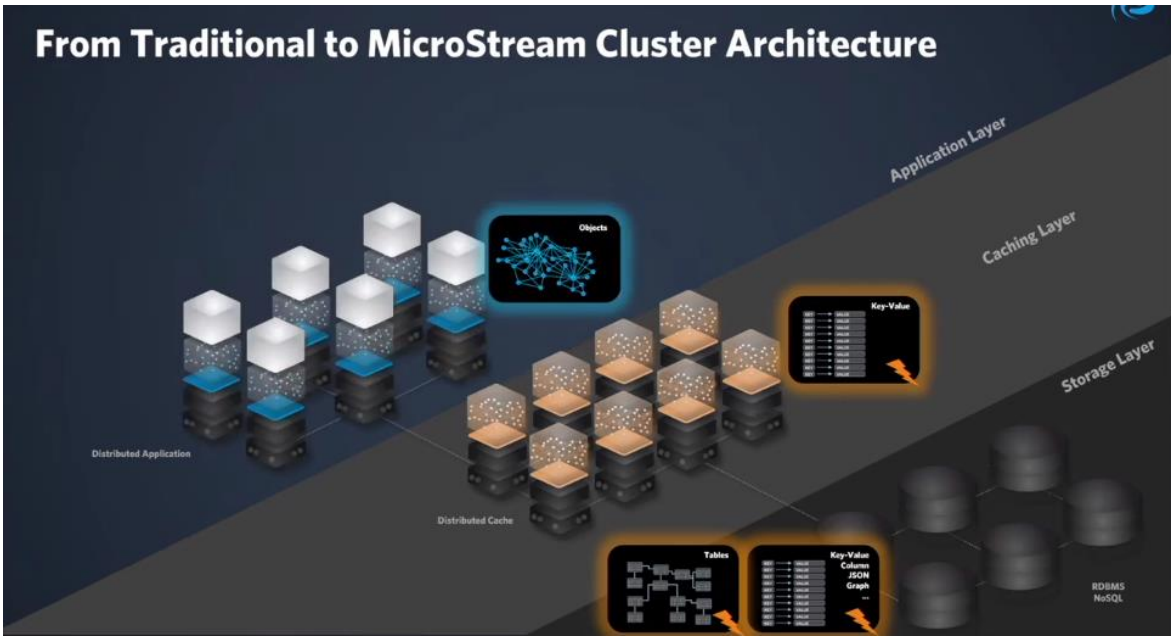

Supported Storages
RDBMS
ORACLE DATABASE
ORACLE TIMESTEN
MySQL
PostgreSQL
MariaDB
amazon DynamoDB
SQLite
NoSQL
mongoDB
redis
kafka
hazelcast
ORACLE Coherence
ORACLE NOSQL DATABASE
Cloud Object Store
amazon S3
Microsoft Azure
ORACLE CLOUD

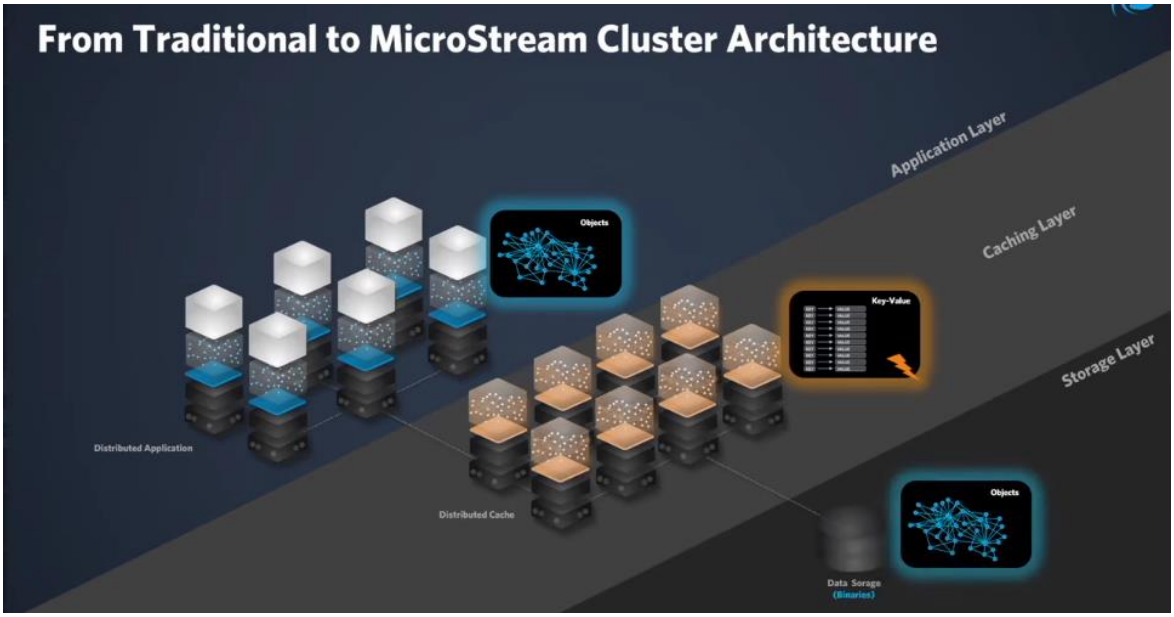

Runs Wherever Java Runs
Desktops
On-Premise
Cloud
Container
GraalVM
Native Image
Microservices
Android
JDK 8+

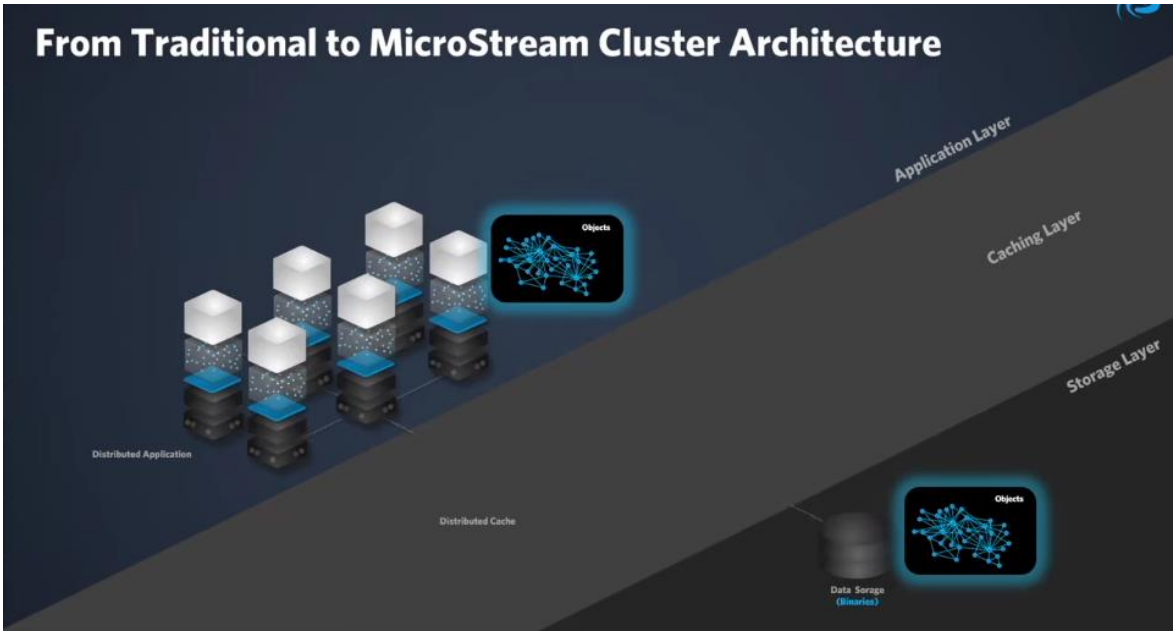

MicroStream Components
MicroStream Storage Browser
REST Interface
Legacy Type Mapping
MicroStream Persistence for Android
MicroStream Persistence for the JVM
MicroStream Serialization
File System Garbage Collector
File System Abstraction
Database Connectors
Backup
Converter (CSV, JSON, Protocol Buffers etc.)
MicroStream Cluster

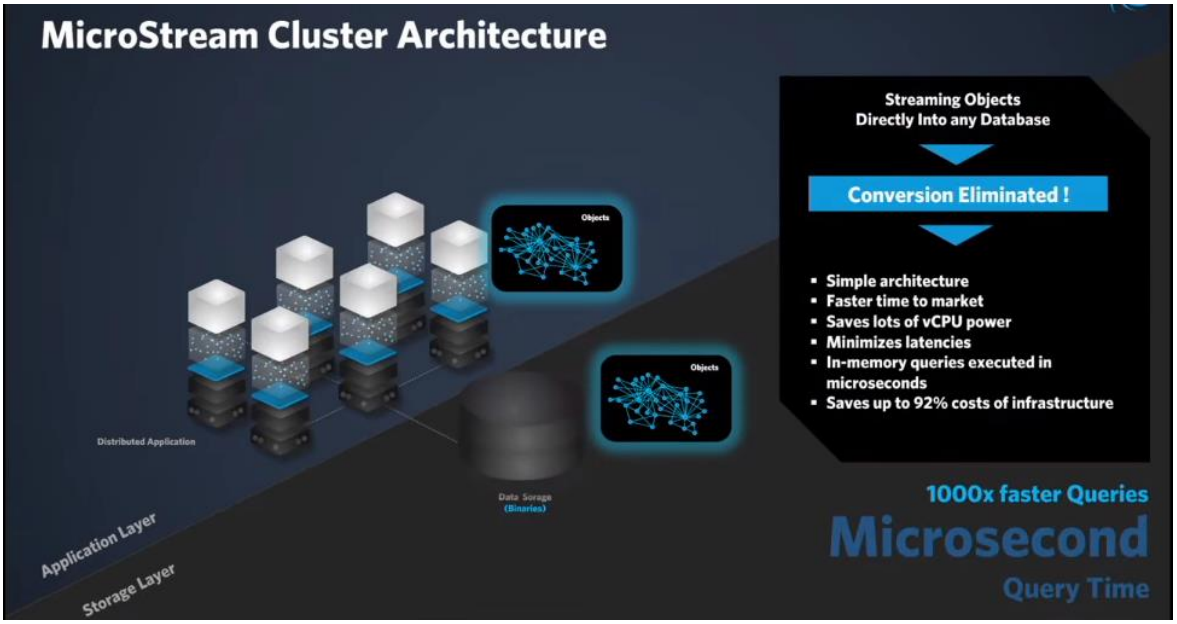
MicroStream Cluster

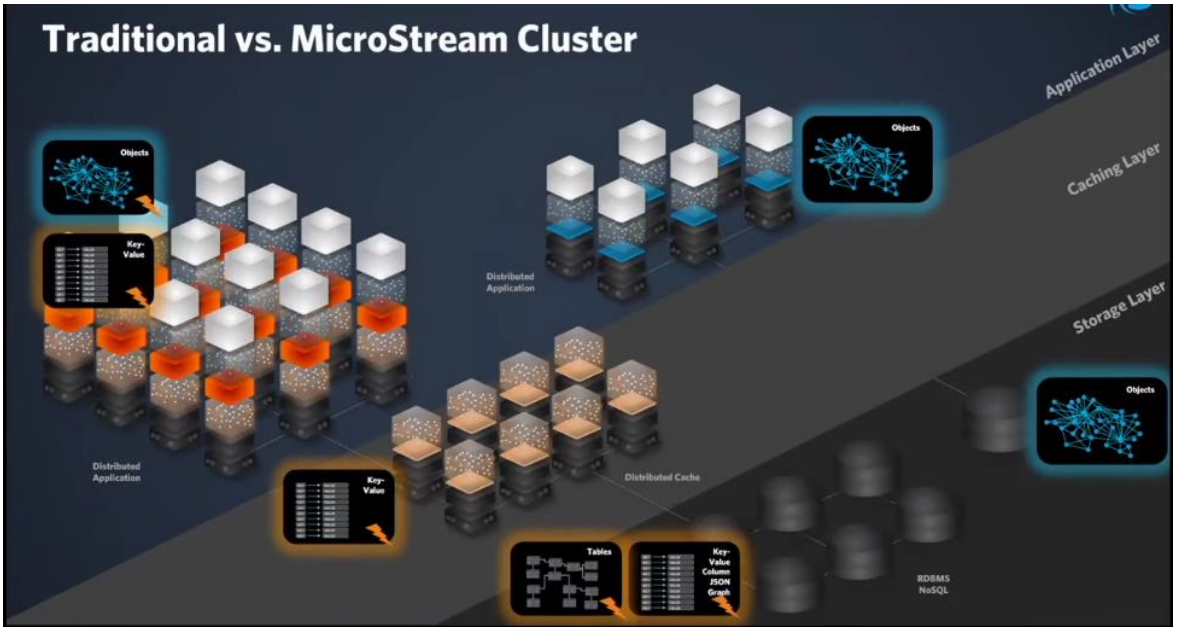


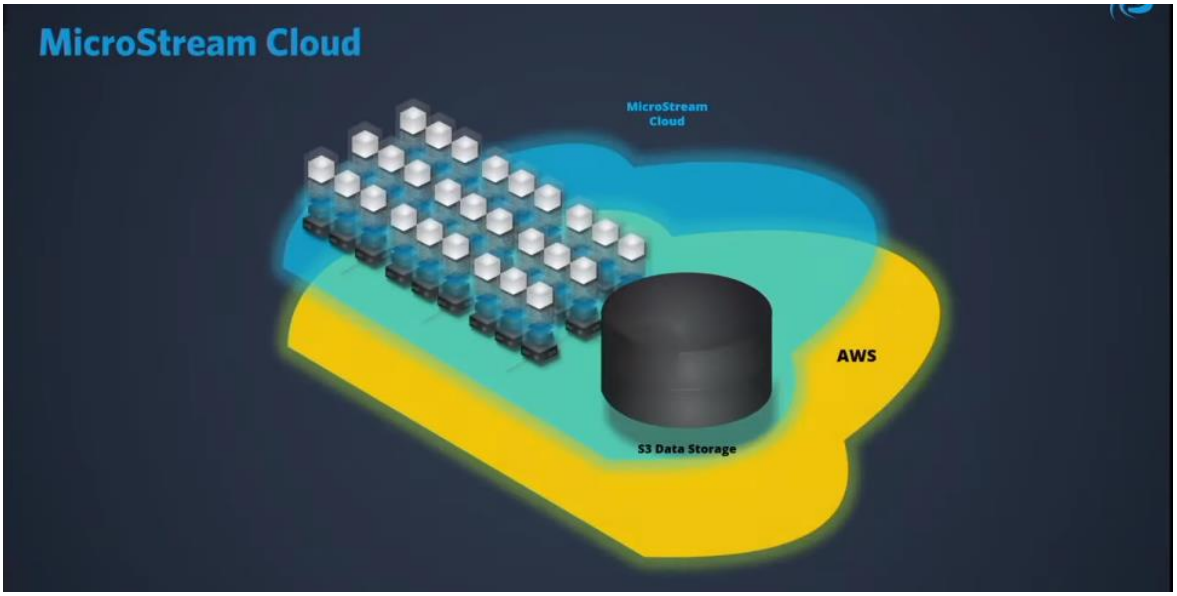

From Traditional to MicroStream Cluster Architecture
Application Layer
Caching Layer
Storage Layer
Objects
Key-Value
Distributed Application
Distributed Cache
Tables
Key-Value
Column
JSON
Graph
...
RDBMS
NoSQL


From Traditional to MicroStream Cluster Architecture
Application Layer
Caching Layer
Storage Layer
Objects
Key-Value
Distributed Application
Distributed Cache
Objects
Data Sorage
(Binaries)


From Traditional to MicroStream Cluster Architecture
Application Layer
Caching Layer
Storage Layer
Objects
Distributed Application
Distributed Cache
Objects
Data Sorage
(Binaries)


MicroStream Cluster Architecture
Streaming Objects
Directly Into any Database
Conversion Eliminated !
Simple architecture
Faster time to market
Saves lots of vCPU power
Minimizes latencies
In-memory queries executed in microseconds
Saves up to 92% costs of infrastructure
Objects
Objects
Distributed Application
Data Soraga
(Binaries)
Application Layer
Storage Layer
1000x faster Queries
Microsecond
Query Time


Traditional vs. MicroStream Cluster
Application Layer
Caching Layer
Storage Layer
Objects
Objects
Key-Value
Distributed Application
Distributed Application
Key-Value
Distributed Cache
Objects
Tables
Key-Value
Column
JSON
Graph
RDBMS
NoSQL


MicroStream Cloud
MicroStream
Cloud
AWS
S3 Data Storage

Benefits - Development

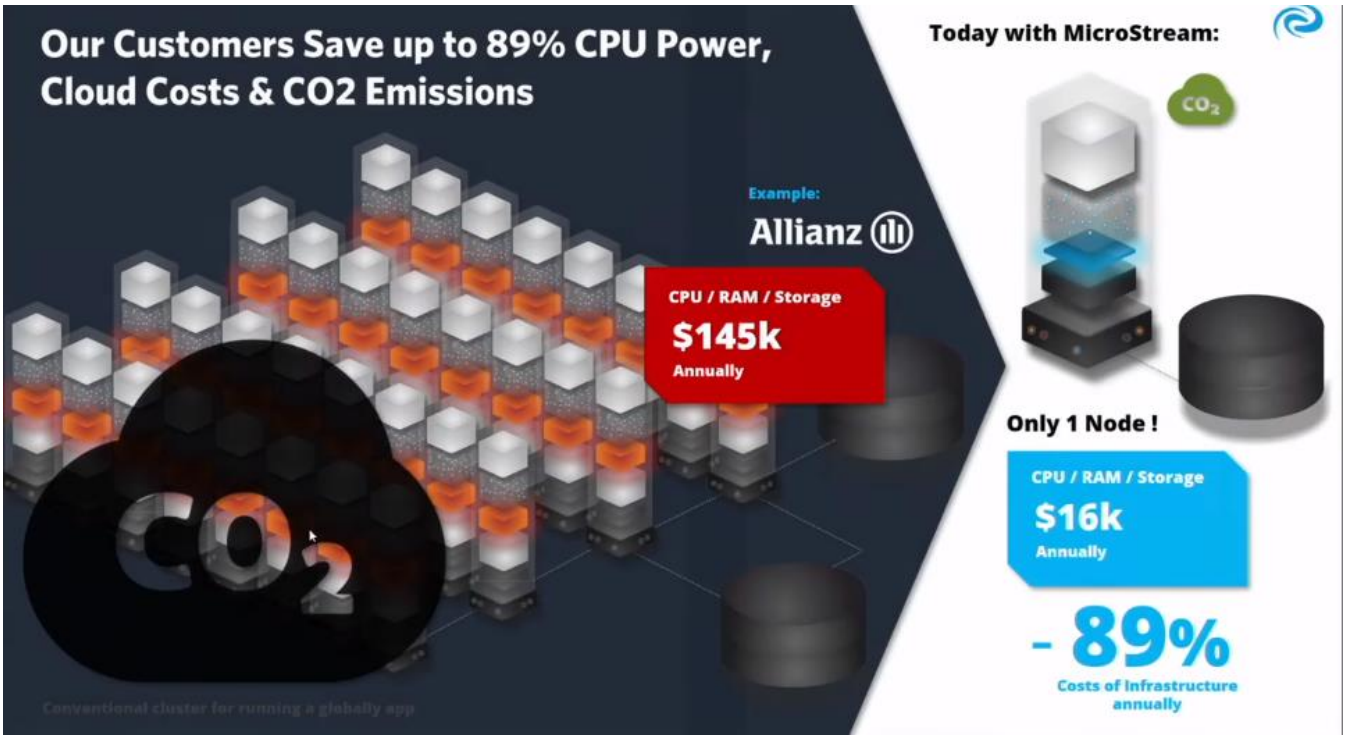
- Simplest possible architecture & implementation
- No mappings or data conversion
- No specific requirements for your classes, just use POJOs
- No specific client API required
- No specific query language
- Core Java concepts only
- Get a clustered app / microservice out-of-the-box

Benefits - Performance

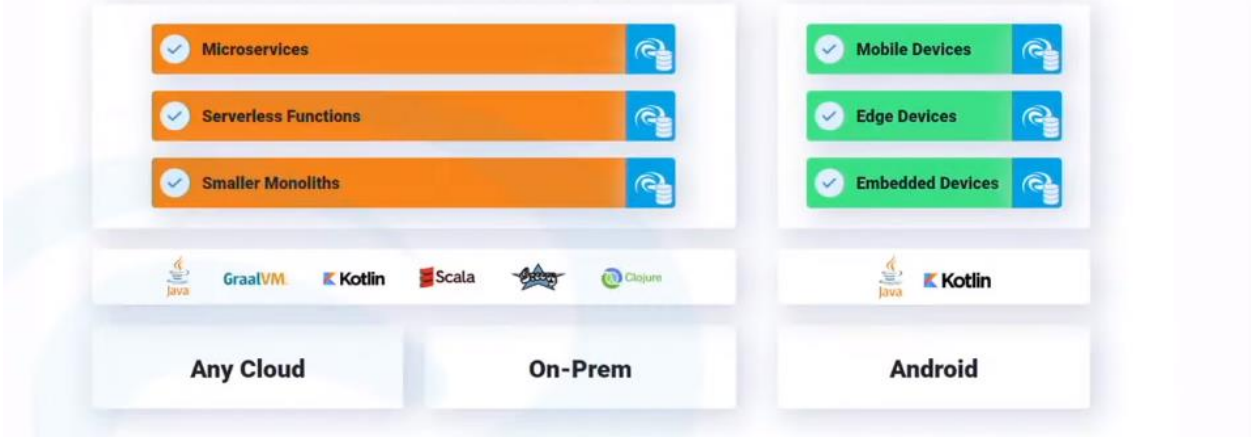
- Ultra-fast in-memory data processing
- Microsecond query time
- Low-latency realtime responsiveness
- Gigantic workloads & throughput

Benefits – Cost Savings

- 90% CPU consumption
- 90% energy
- 90% CO2 emission
- 99% cloud storage costs
- 90% cloud costs in total



Use-Cases

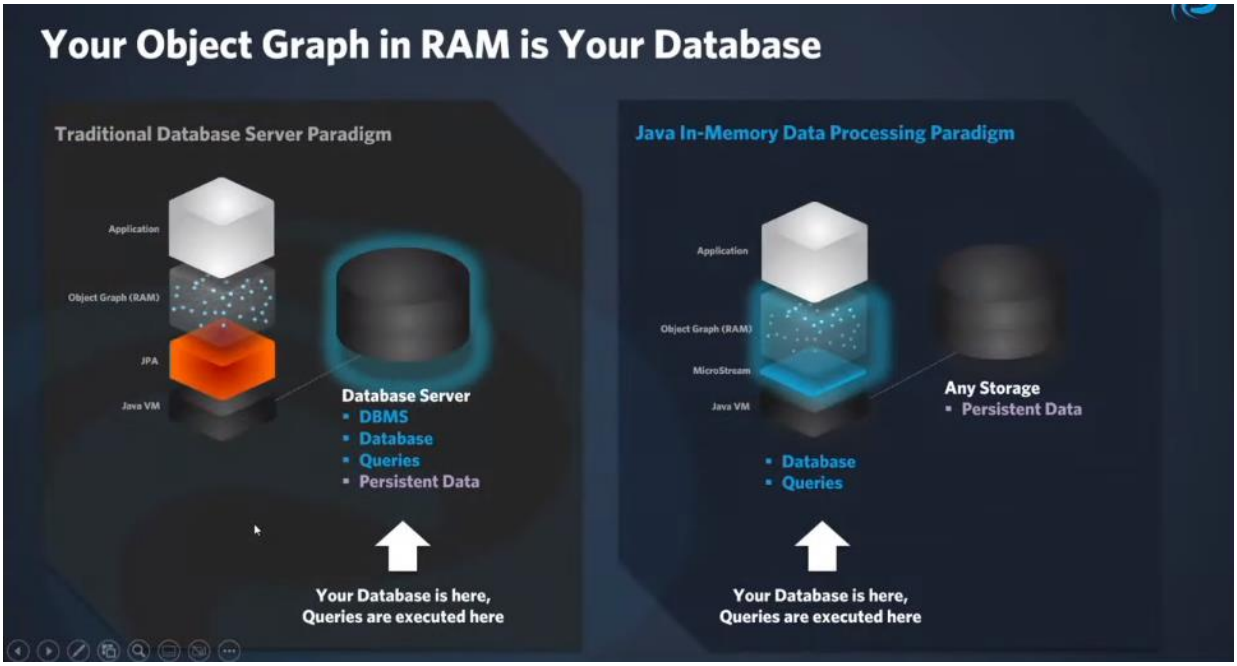


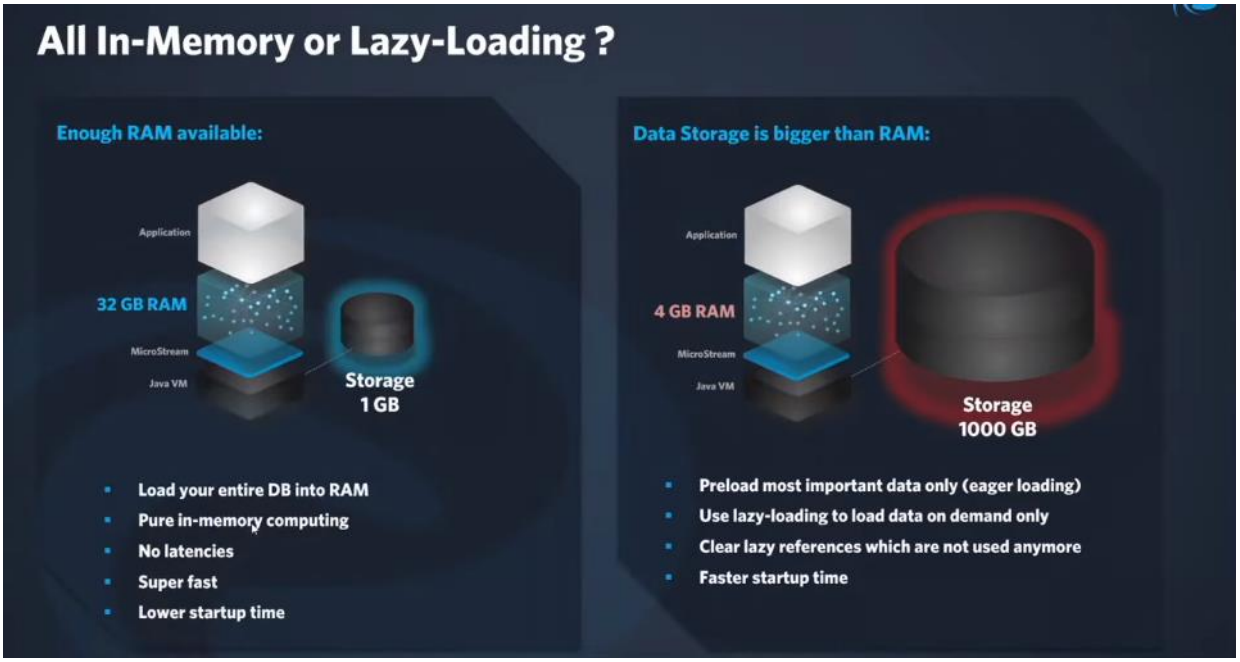
Challenges with MicroStream

- Built for Java developers
- Paradigm shift in database programming
- No SQL support
- MicroStream is a storage engine, but not a DBMS
  - Your application must cover DBMS tasks
  - You must care vor validation
  - You have to care for concurrency
- Not suited for DBAs

Challenges with MicroStream Cluster

- Creating cluster infrastructure is effortful
- DevOps resources required
- Konfiguration von Cluster-Nodes
- Kubernetes know-how required
- Kafka know-how required
- High effort for development & testing
- Monitoring
- No support





Get Started with MicroStream

Learn: www.microstream.one

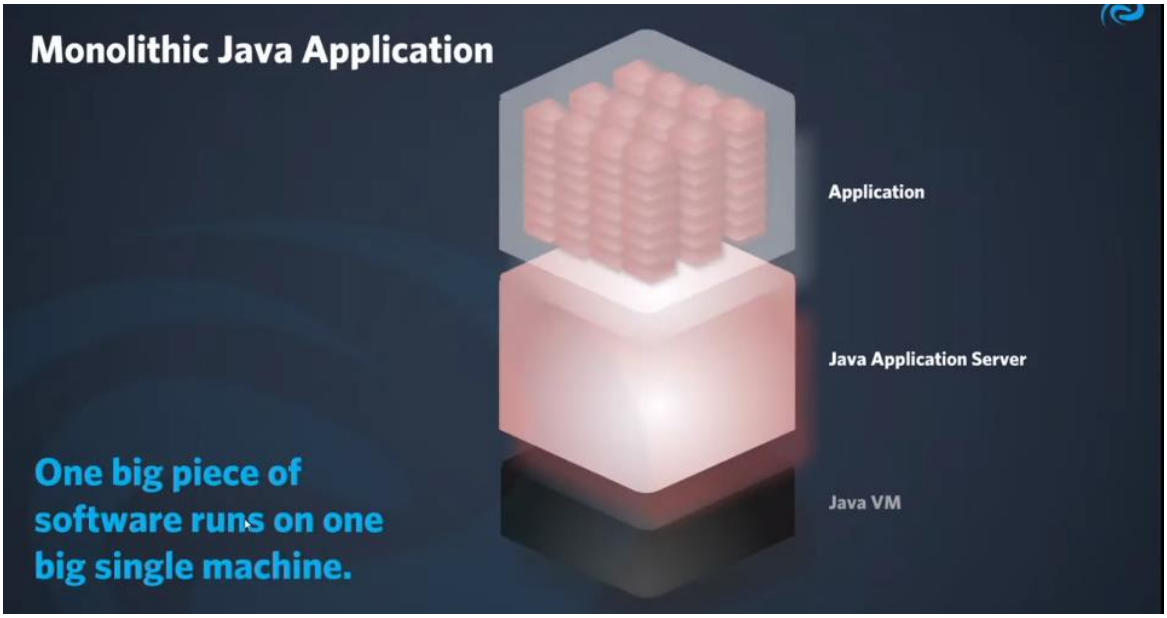
GitHub: https://github.com/microstream-one/microstream

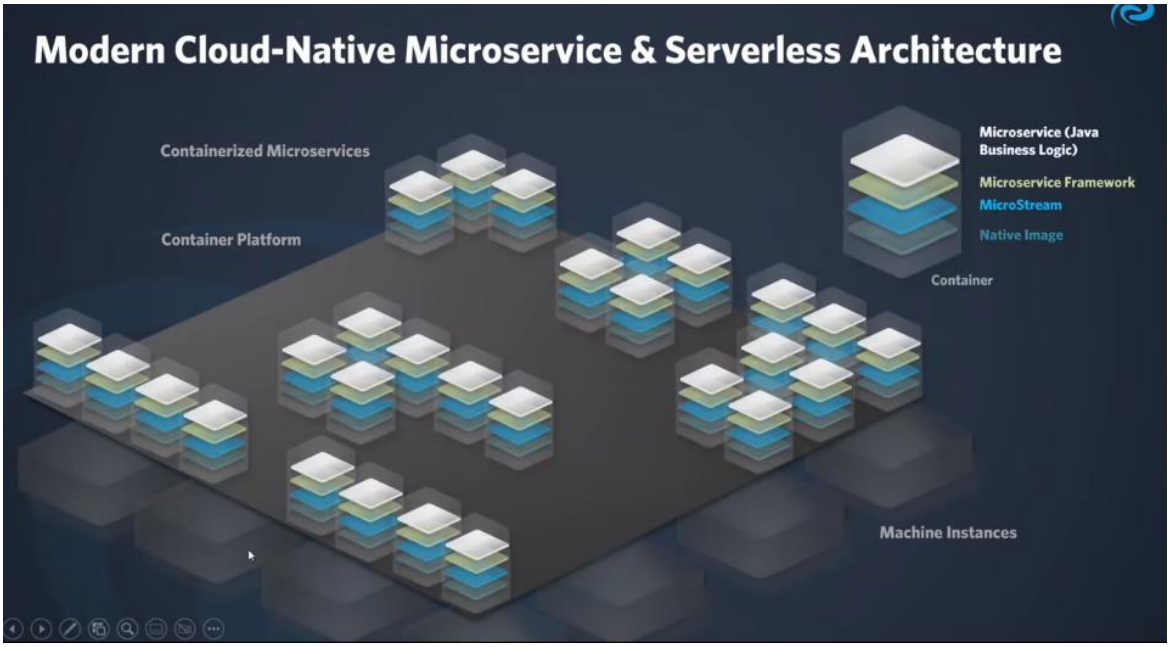
Doc: https://manual.docs.microstream.one/data-store/getting-started

Videos on YouTube: https://www.youtube.com/c/MicroStream/videos

Free MicroStream training: https://www.javapro.io/training

Conclusion


Monolithic Java Application
Application
Java Application Server
Java VM
One big piece of software runs on one big single machine.


Modern Cloud-Native Microservice & Serverless Architecture
Containerized Microservices
Container Platform
Microservice (Java Business Logic)
Microservice Framework
MicroStream
Native Image
Container
Machine Instances


MicroStream


MicroStream