WHITE PAPER

Speedment

Speedment HyperStream

Accelerate Into the Future of Database Applications

Speedment HyperStream

Accelerate Into the Future of Database Applications

The amount of data, the number of transactions and the relationships between data grow more complex and become more dynamic every day. User-demand for agile real-time applications and features is a growing concern for business success. At the same time, numerous companies have valuable data stuck in legacy databases that is difficult to access in a fast and efficient way.

To increase scalability and economic efficiency, organizations migrate their mission critical data to the cloud. However, for a legacy database with myriad interdependencies between the applications and the database or between individual application modules, this is a huge challenge and sometimes not even possible to do.

This paper presents a modern solution for Java applications – Speedment HyperStream, which can increase the read performance from legacy data by orders of magnitude, while making it easy to migrate the whole or parts of the data to the cloud.

The performance increase can be used to bolster customer loyalty, attract new customers, and streamline operations. For a typical relational database application, Speedment HyperStream cuts costs across the board with savings mainly made on avoiding aging systems to consume disproportionately high degrees of resources and maintenance.

Table of Contents

1. The Challenges with Legacy Databases

With rapidly increasing data volumes, new market-demands for more advanced analytics, and evolving application frameworks, a legacy database solution inherently becomes slow and costly to maintain over time. In the following, three major challenges are outlined, which are common in this setting.

PERFORMANCE ISSUES AND HIGH COST TO SCALE

DATA IS LOCKED IN LEGACY SYSTEMS

HIGH COST OF MAINTAINING AND DEVELOPING NEW FEATURES

1.1 Performance Issues and High Cost to Scale

Modern applications put higher demands on the database in terms of latency and bandwidth. Database administrators (DBAs) make continuous efforts to ensure related data is not scattered across several tables and eliminate improper use of primary keys and overuse of stored procedures. DBAs further set out to ensure the indexing and naming conventions follow best practices for database implementation. Despite their efforts, the database performance continues to decline. Companies typically have had their databases for decades and both the amount of data and the complexity of the database are growing every year. Inevitably, the database will run into performance issues, especially for analytical tasks which rely on large amounts of historical data.

The traditional way of scaling a relational database has been to procure larger and often more complex hardware, thereby increasing the processing power, memory, and storage. Although, scaling up the server hardware will at best give a linear performance improvement doubling the computational capability can deliver an effective throughput gain of up to double the bandwidth. While this is often expensive, it may also prove insufficient to keep up with exponentially increasing demands. Therefore, applications typically need a fundamental upgrade to a database solution more suited to the application in question.

Another alternative is to scale out the database by using grid providers that offer in-memory computing solutions where data is located on remote servers or a cluster of servers. This solution provides both database scalability and high performance for certain applications at the cost of affinity, which does not scale out. A database is designed to run on a single server, hence if the data is sharded across several nodes there will be unavoidable delays. Performing a JOIN of tables from different nodes would, for example, constitute a cumbersome task. Another common issue with in-memory solutions is that Garbage Collection within the Java Virtual Machine (JVM) engenders performance obstructions. Garbage Collection is the process that reclaims unused memory. The server is not able to control when the Garbage Collection takes place and when multifold objects live on the heap, thus causing unnecessary delays.

1.2 Data is Locked into Legacy Systems

Organizations routinely contemplate the pros and cons for upgrading their database software to a different engine or migrating the database to the cloud. However, a migration entails not only the costs of assessment, database schema conversion (when changing engines), script conversion, data migration, functional testing, and performance tuning but also fundamental application redesign, all the way from the data model to the application logic. Besides being a complex, multiphase process, there are inherent risks involved; for example, moving data from one database to another is a disruptive operation that may result in data corruption and loss of both service and data if the migration process is not meticulously planned and executed.

Hence, the costs and risks involved with switching to a more powerful or modern database are often deemed too expensive, risky or even implausible for consideration, leaving the database as an unavoidable bottleneck in terms of responsiveness to arising market demands.

1.3 High Cost of Maintaining and Developing New Features

A legacy application using a relational database entails high development and maintenance costs since the codebase of the application is tightly coupled to the database engine interface. Developing new or adjusting existing features that require any change in the data model will also require changes in the code adapting between the database and the application. Therefore, seemingly small changes in the requirements create a need for adapting code at several levels of the system design.

A high cost of maintaining and developing new features does not only constitute a cost per se but does also deter from creating new features due to cost and risk of introducing regression problems. This lock-in effect on an existing set of features is the opposite of the agile characteristics of modern application development.

2. Addressing These Issues with HyperStream

To tackle the three problems described above, Speedment has developed HyperStream. It is an innovative Java tool that enables organizations to get hypersonic application performance without the need for changing or discontinuing the use of the existing database. HyperStream also facilitates migration to the cloud. These benefits are achieved by leveraging the following features:

A UNIQUE **MEMORY MANAGEMENT** MODEL

AUTOMATIC DATA SNAPSHOTS, CODE GENERATION, AND MODULE SYSTEM

A CLEVER **STANDARD JAVA STREAM BASED API**

2.1. Unique Memory Management Model

With Speedment HyperStream, the reading performance can be improved hundreds, or thousands of times compared to reading directly from the legacy database. The key to this achievement is a greatly refined in-memory technology that makes the data directly accessible to the application's CPUs by placing it within the JVM. Speedment's unique memory manager allows data to be stored off-heap to avoid any delays engendered by Java garbage collection operations and as a result gain increased and deterministic performance. Hence, the internal design of HyperStream is carefully planned to take advantage of modern CPU architecture.

Organizations often conclude that they need to implement caching solutions as a means for improving database performance. And, as a consequence, it is not uncommon for companies to invest resources and, put huge efforts into building and integrating their customized cache code. However, these efforts usually prove more challenging than anticipated and introduce steep maintenance costs, thus yielding results that falls short of the expectations. In addition to the great complexity of the caching domain such in-house projects also often undertake several, costly ref refactoring phases during a project's life span. Speedment's main area of expertise is in-memory optimizations. As such HyperStream is built based on vast amounts of accumulated knowledge and expertise within the caching field.

HyperStream relies heavily on memory size for scaling. The current limitation of RAM in modern database computing systems is measured on the order of tens of terabytes, though this is constantly changing as RAM capacities continue to increase and become cheaper year-over-year. In most cases, the RAM size limitation does not prohibit the entire database from being loaded into memory. For organizations with larger amounts of data, Speedment HyperStream offers the option to select specific subsets of the data that will be placed in-JVM-memory. If the application is deployed in the cloud, it is easy to scale up by adding more memory instead of scaling out by adding more nodes. From a cost/performance perspective upgrading the RAM-memory is considerably more efficient than increasing CPU computational capability or using additional nodes.

2.2 Automatic Snapshots, Code Generation and Module System

Valuable data that is locked into a legacy database does not need to be an insurmountable problem. Instead of removing the legacy database, Speedment HyperStream keeps the database as the source of truth. All existing applications that are making writes to the database can be left untouched while new faster applications can be written in a cost-efficient way.

The procedure of reading data from the database and placing it in the in-JVM Memory is done automatically. Speedment HyperStream analyzes the metadata, creates a snapshot of the database and places this close to the application. The organization decides the frequency at which the snapshot is updated, which could, for example, be every hour or every minute, depending on the requirements of the application at hand. All queries made by an application are then directed to the snapshot instead of the database. This means the underlying database is free to focus solely on storing and validating data. Organizations can reap the benefits from an ultra-low latency solution, while preserving the use and integrity of their existing legacy database.

This solution truly facilitates migration to the cloud in a cost-efficient manner and without any risk of losing data. But what about creating micro-services? The primary aim of micro-services architectures is to reject a monolithic application framework. Is it possible to keep a single large database? Yes, Speedment HyperStream has an advanced, built-in ingest system that can be tailored to use any subset of the underlying data, where only specific tables and/or columns are needed for data query and read/write access.Selections can also be made on the data itself; e.g., all sales during this year but not previous years. This enables rapid creation of lightweight micro-services. As an added benefit, Speedment HyperStream supports the Java module system, allowing small micro-service runtimes to be deployed in the cloud.

With automatic snapshots, code generation and system modularization, Speedment HyperStream enables legacy databases to benefit from a range of performance enhancement capabilities, including decoupling of services, miniaturized development, and faster time-to-market for new applications and updates.

2.3. Clever Standard Java APIs

An application dependent on a legacy database typically adopts dated design patterns with high maintenance costs as well as high costs for migrating the code to a more modern framework. The Speedment Stream API provides a modern, future-proof application design based on fundamentals of the Java language and thereby avoids the drawbacks associated with a domain-specific API (read more in a separate white paper on Speedment Stream).

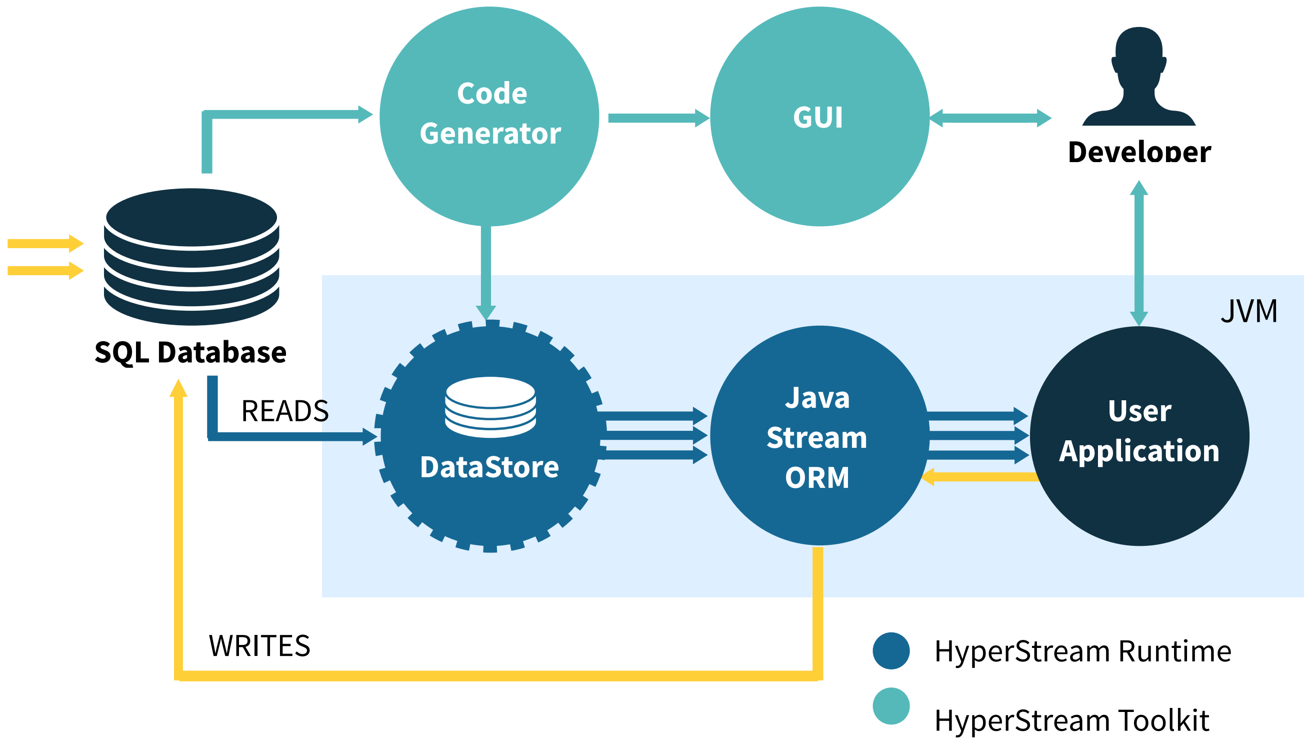
The functional approach to scalable and modular application development has gained momentum as the community of Java developers migrated to Java 8. Widely heralded as the most fundamental step of improvement in many years, this update modernized the language to step into the realm of functional programming, allowing for more efficient, clear and elegant code that therefore is fundamentally less costly to maintain. The HyperStream API enables the developer to access a database using pure Java Streams. Given that no specialized interface is needed, adopting the business logic for use with HyperStream only amounts to future-proofing the application by leveraging the improvements of the Java language itself.

An API that is standards compliant and uses only constructs native to the Java language gives a future-proof application that allows for developing new features and cost-efficient maintenance. It also simplifies the process of finding qualified developers, as most Java developers are proficient in the use of the Java Stream API.

User applications may use two different APIs, which depends entirely on whether the application is server- or client-side. For a client-side applications, Speedment HyperStream can deliver the data over a REST API, while server-side applications coexisting with the runtime uses the Stream-based API described above.

3. HyperStream Architecture

Speedment HyperStream is an extension of Speedment Stream, meaning HyperStream's abilities go beyond efficient development of Java database applications and includes far more advanced runtime properties. Like Stream, HyperStream includes a Toolkit and a Runtime. The Toolkit generates code that encapsulates specific database contents and the Runtime provides the data handling functionality for the user application.



The Speedment HyperStream architecture is composed of a Toolkit and a Runtime that interacts with the underlying database and the user application.

3.1 Toolkit

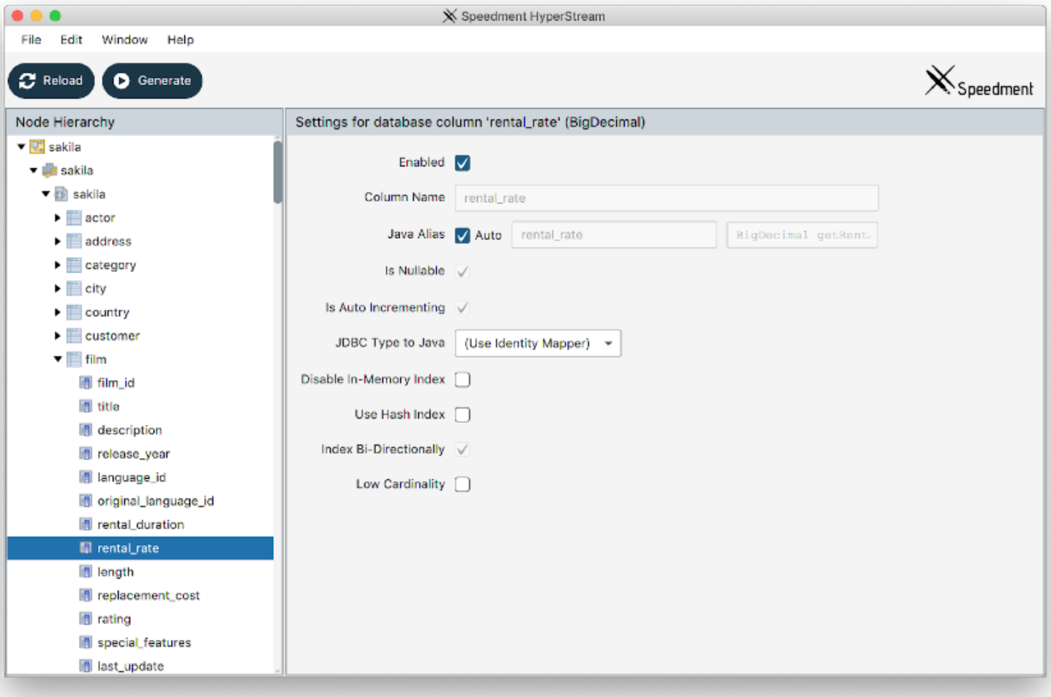
The Toolkit contains a Code Generator and a Graphical User Interface (GUI) that greatly facilitates creation and maintenance of database applications.

3.1.1 Code Generator

The Code Generator analyses the underlying data sources' metadata and automatically generates a full Java domain model. This eliminates the tedious task of writing boilerplate code such as Java Entity Classes. The Code Generator also allows seamless synchronization of schema changes between the database and the application by enabling simple regeneration of the domain model at any time.

3.1.2 Graphical User Interface (GUI)

With the Speedment HyperStream graphic user interface, developers can control the code generation process and perform a range of custom configurations. Once ready with the configurations, all it takes is a click on "Generate" button and the Java domain model is automatically generated in an instant. Any custom configurations are saved for future regenerations.



Settings for database column 'rental_rate' (BigDecimal)

Enabled	✓
Column Name	rental_rate
Java Alias	✓ Auto rental_rate BigDecimal getRent…
Is Nullable	✓
Is Auto Incrementing	✓
JDBC Type to Java	(Use Identity Mapper)
Disable In-Memory Index	☐
Use Hash Index	☐
Index Bi-Directionally	✓
Low Cardinality	☐

The Speedment HyperStream graphical user interface, which enables developers to control and configure the code generation process.

3.2 Runtime

The Runtime provides the data handling functionality and consists mainly of a Java Stream Object Relational Mapper (ORM) and the memory manager DataStore. These components are placed within the JVM itself, close to the user application, to ensure minimal latency.

3.2.1 Java Stream ORM

Instead of typing SQL or using a third-party API, queries are expressed using standard Java Streams. As a standard Java Interface, streams do not include any details about how data is retrieved. Instead, this is delegated to the framework defining the pipeline source and termination. Speedment HyperStream leverages the Java Streams architecture to establish a connection with a snapshot of the database stored in RAM rather than a remote database. With this unique in-JVM-memory approach, Java Streams execute queries that are orders of magnitude faster. This constitutes a new way to write highly performant data applications, whereby an existing database remains the source-of-truth. As an added benefit, developers are equipped with an expressive and type-safe language that minimizes the risk for introducing bugs into application code.

3.2.2 DataStore

Database entities are stored in-JVM-memory, allowing database queries to be executed blazingly fast, utilizing the Java Stream API to the fullest extent. Streams can have a latency well under one microsecond (μs). This can be compared to a traditional database connection, where the TCP round-trip delay, even in high-performance networks, is rarely under 40ms. Additional concerns include database latency and data transfer time. DataStore's abilities are tailor-made for read-intensive applications, such as data analytics.

Having established a broad picture of how DataStore operates, the following paragraphs describe several properties of DataStore in greater detail to explain the exceptional performance.

Data is Stored Off-Heap

HyperStream's use of Java entity classes and streams might appear to result in vast amounts of Java Objects stored on the heap; however, this is not the case. Heap-allocation is associated with increasingly large CPU requirements for Java Garbage Collection, often causing prohibitive delays if the heap becomes too large. The overhead for heap objects also becomes significant as the standard Java Map and TreeMap constructs create an abundance of excessive support objects. A single Java entity object with twenty columns can easily create a hundred times as many support objects when indexing maps are taken into consideration. This is unacceptable in low-latency scenarios. Rather, to handle vast amounts of data within the JVM, DataStore is designed to manage data off-heap.

DataStore serializes Java entity objects in a compact binary format off-heap without leaving a single object unaccounted for. Here, ByteBuffers are used as the fundamental memory regions, without the occurrence of Java Unsafe. The same is true for indexes which also live completely off-heap. This paves the way for JVMs operating in terabytes of RAM with a deterministic performance regardless of size, limited only by the available RAM.

Partial Deserialization of Entities

Another salient aspect of HyperStream's performant data retrieval is its ability to partially deserialize entities. If, for example one was to compute the average age of person entities with a hundred columns, it would be wasteful to restore a complete Person object from the memory to read a single value (the age). Instead, HyperStream decomposes the averaging stream and obtains an accessor that can read the relevant value directly from the serialized form in RAM, thereby avoiding deserialization of the entire entity.

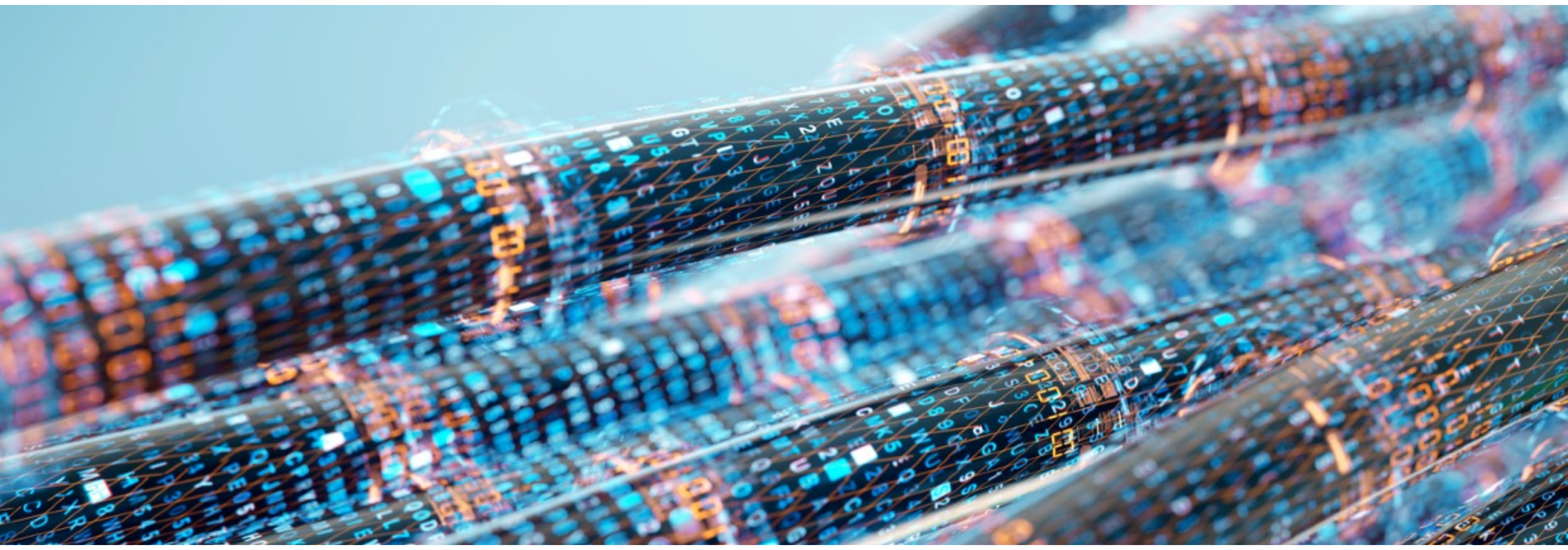
Terabytes of Data

The built-in data structures of Java, such as a HashMap, perform well for small and medium data sizes. For larger data sets, the performance of these data structures deteriorates directly as a function of data size. The use of these built-in data structures is also limited to holding two billion entities, which is the maximum array size in Java using 31-bit addressing. In contrast, the HyperStream's DataStore has a memory manager that uses a 55-bit addressing scheme, which is pointing directly to entities serialized in off-heap memory. This allows the handling of over one hundred trillion entities per table for entities with an average size of 128 bytes.

Lock-free and Thread-safe

In previous decades, modern computing hardware has scaled mainly in terms of the number of computational units (CPU cores), whereby each core has gained only a moderate performance increase during the last iterations. Today, CPUs with 64 cores and 128 threads have become the norm for database server systems. Because of this, the HyperStream's DataStore relies on an immutable and, therefore, an inherently lock-free and thread-safe storage engine that allows thousands of simultaneous threads to access data. Both entity and index stores are hierarchically organized to allow the reuse of most data laid out in memory between database snapshot generations.

With a lock-free and concurrent architecture, HyperStream's DataStore is highly scalable. Paired with its ultra-low latency capacity, this property inherently provides an impressive query capacity. For example, over a hundred million Java streams can be created and consumed per second on commercially available servers and cloud instances. This means most applications



meet the required service-level with margin. Seen the other way around, even a modest server or tiny cloud instance can provide millions of Java streams per second paving the way for efficient micro-service instances in the cloud. This is important for organizations that deploy instances in the hundreds or even in the thousands to achieve a cost-efficient infrastructure.

Dynamic Joins of In-JVM-Memory Tables

HyperStream's DataStore supports dynamic joining of up to ten in-JVM-memory tables, thereby saving valuable space compared to storing denormalized forms of the data. The join engine supports all commonly used join operations, such as Inner Join, Left Join, Right Join, and Cross Join. As mentioned above, DataStore can decompose streams over a single table and extract only the specific columns used. This is also true for dynamically joined tables whereby the join function as such may be decomposed to reveal only the used column accessors for the different columns referenced in the join operation. This saves execution time and eliminates intermediary object creation.

Efficient Off-heap Aggregations

For analytical use cases, it is common to create aggregations over data sets; e.g., a table or dynamically created joined tables. Such operations are handled by the HyperStream Aggregator, which, like DataStore, operates completely off-heap by decomposing streams and obtaining the relevant column extractors that interact with data stored directly in RAM. All internal temporary aggregations are created off-heap, thereby avoiding any intermediary object creation. Moreover, the Aggregator supports parallel lock-free aggregations, whereby each thread accesses a local aggregation. These aggregations are subsequently combined upon thread completion.

Lightweight In-JVM-Memory indexes

Data is filtered using one or more predicates applied to a set of columns. For example, an application might query and access sales statistics from the previous month. To accommodate a fast selection of data subsets and rapid sorting of this data, HyperStream's DataStore addresses columns with a specialized, light-weighted index with minimal memory overhead, allowing efficient filtering and sorting on an ad hoc basis. The indexes are also stored off-heap, creating a scalable DataStore. Hence, all index operations are largely independent of how much data is stored. Therefore, comparisons such as equals, greaterThan, lessOrEquals, between, in, sort, etc. have a time complexity of O(1) or O(log(N)), leaving search-latency unaffected by large datasets.

Decomposition of Streams and Virtual Placeholders in the Code Logic

The internal design of HyperStream's DataStore is carefully planned to take advantage of modern CPU cores with L1, L2, and L3 caches. Latencies are kept to a minimum by decomposing streams in its constituent parts to calculate internally the most efficient execution path. Even though objects appear to exist in a stream definition, they can often be optimized away and exist only as a virtual placeholder in the code logic. The actual execution plan may bypass object creation and can produce data streams directly from RAM. .

Loading of Snapshots in the Background

Even though HyperStream's DataStore is geared towards data analytics, it still provides a means to update its internal data representation. DataStore may load an updated snapshot in the background while serving existing threads. Once loaded, new threads/transactions will query against the new snapshot, whereas old queries will be run against the old snapshot. The snapshot reload time can be set programmatically; e.g., each day, every hour, every five minutes, etc.. The DataStore also supports incremental snapshots, whereby a new snapshot depends on the previous one. Incremental snapshots are low-cost in terms of CPU and memory resources, and can be made several times per second. An incremental snapshot recycles most of the immutable data structures from a preceding snapshot

3.3 Modularized for Light-weight Deployments

HyperStream supports the Java Platform Module System (JPMS) and, by doing so, benefits from continuous improvements in JVMs, including Java 11+ and GraalVM. JPMS allows tailor-made Java runtimes to be distilled, whereby only the required modules are incorporated into the runtime, greatly reducing storage and memory requirements for the application. Use of the JPMS feature is optional and, by using multi-version jars, HyperStream can run on Java 8 and subsequent Java releases.

3.4 Hardware and Software Independent

HyperStream relies only on standard Java interfaces and does not require any special hardware or platform-specific software. Thus, without restriction, HyperStream can run on any Java platform, ranging from small ARM processors to massive supercomputers. Deployments can be made using any cloud provider and technology, including AWS, Google, Azur, Docker, and OpenShift, on-premises or in private clouds.

4. Conclusion

Speedment HyperStream enables organizations with a demand for low-latency database access to build scalable high-performant systems for agile real-time market demands. Additionally, companies with a software monolith, currently unable to meet modern efficiency demands, can leverage HyperStream to reuse a legacy database without the need for risky and costly database switches and obtain a modern database architecture that performs better than most NoSQL solutions. With Speedment HyperStream, modern database architectures need not be restricted to a particular ecosystem and provide elastic scalability, fast creation, and deployment of new features, as well as easy maintenance and transparency in the auto-generated code.

Speedment, Inc.
470 Ramona Street
Palo Alto 94301, CA, USA

info@speedment.com