Independent degree project – second cycle

Master’s thesis MA, 30 credits

Master of Science in Computer Engineering

Parallelism within queue application

Ludvig Åberg
Abstract

The aim of this thesis was to modify an existing order queue application which was unable to execute orders in a queue in parallel which in turn could lead to a bad user experience due to the increased queue delay. The thesis proposes two queue structures to allow parallel execution within a queue. One of the two is selected for implemented in the modified order queue application. The implementation was carried out in Java EE and used different types of frameworks such as JPQL. Some parts of the order queue application had to be modified to be able to handle the new queue structure. New attributes that defines dependencies of the orders are used to find a suitable parent for each order in the queue. The queue structure was visualized making it possible to see the execution in real time, and a test server was implemented to test the queue structure. This resulted in a working prototype able to handle dependencies and parallel orders. The modified order queue application was performance measured and compared to the original order queue application. The measurement showed that the modified order queue application performed better than the original order queue application in terms of execution time below a certain number of queues. Future work includes optimizing the methods and queries in the implementation to increase the performance and to handle parallelism within the orders.

Keywords: Java EE, parallel order queue, transaction, JPQL
Acknowledgements

I would like to thank those who have helped me with this thesis:

My supervisor Magnus Eriksson at Mid Sweden University Sundsvall, who have given me valuable help with the thesis.

My supervisor Lennart Bohman at Dewire who has given me valuable insight regarding the problem I faced and the result I produced.
Table of Contents

Abstract iii
Acknowledgements iv
Introduction 1
Background and problem motivation 1
Overall aim 2
Scope 2
Concrete and verifiable goals 2
Outline 3
Contributions 3
Theory 4
This chapter presents the theory and frameworks used in the thesis. 4
Transaction 4
Nested Transactions 5
Scheduling 5
Threads 6
Frameworks and data types 7
JPA 7
D3.js 7
RESTeasy 7
Java EE 7
REST 8
JSON 8
Database 8
Database management system 8
MySQL 8
Structured Query Language 9
Entity Relationship Model 9
Queries 9
Index 10
Order Engine 11
Related work 12
Methodology 15
Research 15
Performance evaluation 15
Implementation 15
Implementation 16
Queue structure proposal 1 16
Insertion 17
Deletion 18
Finding subtrees 20
Queue structure proposal 2 21
Insertion 22
Finding nodes 24
Deletion 25
Parallel orders implementation 27
Parallelism within queue application
Ludvig Åberg

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Visualization</td>
<td>29</td>
</tr>
<tr>
<td>Test server implementation</td>
<td>30</td>
</tr>
<tr>
<td>Results</td>
<td>33</td>
</tr>
<tr>
<td>Execution time per queue</td>
<td>33</td>
</tr>
<tr>
<td>Orders per second</td>
<td>35</td>
</tr>
<tr>
<td>Insert order</td>
<td>37</td>
</tr>
<tr>
<td>Insert into queue</td>
<td>40</td>
</tr>
<tr>
<td>Response time</td>
<td>42</td>
</tr>
<tr>
<td>Scheduler fetch time</td>
<td>44</td>
</tr>
<tr>
<td>Evaluation</td>
<td>47</td>
</tr>
<tr>
<td>Conclusions</td>
<td>48</td>
</tr>
<tr>
<td>Ethical discussion</td>
<td>49</td>
</tr>
<tr>
<td>Future work</td>
<td>49</td>
</tr>
<tr>
<td>References</td>
<td>50</td>
</tr>
</tbody>
</table>
Parallelism within queue application
Ludvig Åberg
2017-06-02

1 Introduction

Today there are many different types of queues in our society. For example we have car queues and grocery queues, which could be considered as first in first out queues. If there is only one car queue or grocery queue, it will take a long time for the driver or customer to get through, if there are many people in line. Being stuck in a queue for too long could be frustrating. But by adding additional car lanes or multiple grocery checkouts the queue can handle cars or customers in parallel, thus increasing the throughput of the queue which in the end will mean that the people in the queue will get out faster. This also applies to software applications: if a user is stuck at a loading screen too long they could get frustrated and have a bad experience.

This thesis proposes a modification of an existing order queue application order engine developed by Dewire to allow parallelism within the queues.

This chapter will describe the background of the thesis and the problem it aims to solve.

1.1 Background and problem motivation

Dewire is an IT consultancy company specializing in mobile solutions that develop and maintain IT-systems specified by customer need. Dewire has developed an order queue which handles transactions made in their customers’ IT systems. When the user transaction is made it is placed in a specific queue in the order queue application. If this queue has many transactions waiting for execution, the transaction may be delayed before it can be executed. This results in a bad user experience because of the time it takes for the transaction to be completed.

The current implementation executes transactions sequential from each existing queue, but since the queues are isolated from each other it executes transactions from different queues in parallel. The current implementation can execute up to 10 transactions at the same time provided that no transactions share the same queue. If there is only one active queue in the order queue application it will only execute transactions one at a time. This is a huge performance loss and contributes to queue delays.

This thesis proposes modifications to the order queue application to be able to execute transactions in parallel within the same queue whenever possible. This could increase the performance in certain situations such as having less than 10 active queues in the system. These modifications could prove to increase the throughput of orders and reduce the queue delays for users.
Parallelism within queue application  
Ludvig Åberg  
2017-06-02

To measure the proposed modifications an application was created to generate dummy transactions with certain properties and send these to the modified order queue application.

1.2 Overall aim

The overall aim of this thesis have been to propose modifications to an existing application. This includes of proposing a new queue structure that supports a parallel transaction execution which is as efficient as possible as well as modifications in order queue application to be able to use the parallel support. The purpose was to increase the performance by allowing transactions within a queue to be executed in parallel. Measuring the performance and comparing to the original order queue application will show if the proposed modifications have increased the performance of the order queue application.

1.3 Scope

The performance measurement will only be conducted up to 14 queues since measuring the order queue application is time-consuming and the number of threads that is able to execute transactions are 10.

The security aspect has not been considered and can be applied on the end product if necessary.

1.4 Concrete and verifiable goals

- How can a queue structure support parallel transaction execution with respect to transaction dependencies?

- How can this be implemented, integrated and visualized into the existing order queue application?

- How will the proposed queue structure perform in terms of load and throughput in regards to:
  - Execution time;
  - Order per second;
  - Order insertion time;
  - Queue insertion time;
  - Response time;
  - Scheduler fetch time?
1.5 Outline

Chapter 2 describes the theory of the thesis.

Chapter 3 explains the methods used for the implementations and results.

Chapter 4 describes the proposed queue structures and implementation of the applications.

Chapter 5 presents the results of the measurements of the modified order queue application.

Chapter 6 analyzes the recorded results.

Chapter 7 concludes the thesis.

1.6 Contributions

The thesis idea was provided by Dewire with a description of the current problem. Dewire also provided the original order queue application upon which modifications were applied.
2 Theory

This chapter presents the theory and frameworks used in the thesis.

2.1 Transaction

A transaction is defined as a set of tasks that updates a database in a logical ordering; the tasks can be either a read or write operation. The transaction succeeds only if all tasks included are succeeded. If all the tasks of a transaction succeed the transaction will commit to the database and complete, if one task or more fails the transaction will fail and start a rollback which will undo all the succeeded tasks to the previous state. A transaction can be generated by a user or automatically by a computer. [1][20]

A transaction always have four properties which guarantees “safe” data in the database. The synonym is ACID which stands for Atomic, Consistency, Isolation and Durability. [20]

• Atomic; means that the transaction is completed as a unit; if it does not complete it does not affect the system. If a transaction does not complete it must ensure that all changes made by the transaction are undone and restored to its previous state. Atomic execution implies that each transaction is either completed or aborted. A transaction can be aborted for several reasons: system crashes during execution, the transaction being involved in a deadlock and aborted, the transaction violating an integrity constraint and therefore aborted.

• Consistency; means that each transaction will bring the database from one valid state to another. I.e, all data that have been written to the database follow the rules defined for the database. These rules could includes constrains, cascades and triggers. The database must satisfy all integrity constraints, i.e. the execution of each transaction must maintain all integrity constraints.

• Isolation; means that concurrent executions result in a database state that would occur if the executions were made in serialization. If a transaction fails, the effect will not be visible to other transactions.

• Durability; means that the transaction that has been committed to the database will be stored regardless of power failure, computer crash or errors. Durability can be achieved by redundant data storage on different backup devices. In the real world, there are some types of events that can threaten the durability of the transaction processing system, such as CPU crash, disk failure, fire, and malicious attacks.
Parallelism within queue application
Ludvig Åberg
2017-06-02

Rollback
Rollback is a method used in a transaction database which ensures the integrity of the database. It saves a copy of the data in the database prior to a transaction, and if the transaction fails in some way, a rollback is used to restore the data to a consistent state. [1][20]

Deadlocks
Deadlocks occur if two or more transactions access the same data from the database at the same time. Deadlocks will prevent execution since both the transactions will wait for other. Deadlocks are prevented by detection. When a deadlock is detected both the transactions are canceled and rolled back.[1][20]

2.1.1 Nested Transactions
Nested transactions is a transaction which has one or more child transactions, where the depth of the children is arbitrary. The parent transaction cannot execute its operation until every child transaction has either committed or aborted, however the parent transaction can abort. The child transaction execute atomically with respect to their siblings and after the child transaction commits the changes are then seen by their parent transaction but not made permanent. The changes are made permanent after the parent transaction commits. [2][3]

2.2 Scheduling
Scheduling is a method which has specified rules how to assign tasks to certain resources to complete the task. The resources could be a processor or network card. A scheduler wants to keep the resources busy if possible, and allow users to access the same system resources effectively. A scheduler may work to improve different performance issues depending on what is needed in the system in question. It may improve the throughput of the system, which is the total amount of tasks executed within a time frame. It may minimize the response time, i.e. how long it takes to execute it from the point when it was enabled. It could also minimize the delay, that is, how long it takes to complete subsequent tasks. It could also aim to increase fairness which means giving all tasks equal process time.

The scheduler could choose tasks from the task queue depending on different variables. It may choose tasks based on priorities defined by the system. A task with high priority needs to be executed fast as it may consist of tasks that are critical to the system, or other tasks in the queue. The scheduler can choose tasks depending on the number of resources the tasks require to for execution,
Parallelism within queue application
Ludvig Åberg 2017-06-02
and it may choose tasks depending on the dependencies of the tasks, which may increase the effectiveness by allowing tasks to be executed in parallel. [22]

2.3 Threads

Threads are created in a processor sharing the same memory, whereas processors do not. Threads created in a software application enable parallel execution. Having only one thread in an application means that single thread will execute everything within the application. By having more threads, two executions can be done in parallel.[23]

By using threads, an application may increase responsiveness, which means that if there are tasks within an application requiring a large amount of time, everything in the application will be blocked until the task has been completed, but using 2 or more threads the other threads can handle other tasks or inputs, while the time consuming task is being processed. It also enables faster execution because it allows parallel execution. By using threads it also increases the use of the system, because more tasks can be handled at the same time.

A drawback of using threads is that it must be ensured that no threads are working on the same critical section at the same time, because it will result in corrupted or incorrect data. Mutual exclusion can be used to ensure that only one thread works at a time in a critical section. Another drawback is that if a thread carries out an illegal operation and crashes the whole process, the threads belonging to the process crashes as well. [22]

Thread safety is needed in multi threaded programs to ensure safe execution by multiple threads at the same time.
2.4 Frameworks and data types

2.4.1 JPA

Java Persistence API (JPA) is a Java programming interface that can be used to describe the management of data in a relational database. JPA is used to simplify the connection between Java and a database. It maps Java objects with relational models. To manage the tables in the relational database an entity is created in a way similar to a Java class. An entity is mapped to individual rows in the database and may have a relationship with other entities which is declared by using Java annotations.

It uses Java Persistence Query Language (JPQL) and is used to query the entities mapped to a relational database. It has the same syntax as SQL queries, but only communicates with the entities, never with the database. When a JPQL query is executed it returns an entity object. However, it is limited in its functions compared to SQL since it does not support all functions available in SQL. [4][5][6][7][20]

2.4.2 D3.js

D3.js is a Javascript data visualization library which enables creation of dynamic and interactive data visualization in a web browser using HTML, SVG and CSS. [8]

2.4.3 RESTeasy

RESTeasy is a Jboss implementation of Java API for RESTful web services and follows the REST architectural pattern. It uses annotations to specify the properties of the web service. The annotations that can be used are: [9]

- @Path; specifies the path for the web resources,
- @GET / @PUT / @POST / @DELETE / @HEAD; specifies the http request type of the web resource,
- @PRODUCES; states the type of response the web resource sends,
- @CONSUMES; states the type of request the web resource accepts.

2.4.4 Java EE

Java Enterprise Edition (Java EE) is used to develop enterprise software and uses the object oriented Java programming language.
2.4.5 REST

Representational State Transfer (REST) is a standard that specifies how to allow machine to machine communication. It enables access and modifications using HTTP protocol operations such as GET, POST, PUT and DELETE. The HTTP protocol is stateless; each request between a client and a server must contain all of the information necessary to understand the request [rest arch]. The HTTP methods: [10][11]

- GET is used to retrieve information from the server such as information about employees or information about a specific employee at a company,

- POST is used to create new entries on the server, such as an entry for a new employee at a company,

- PUT is used to update an entry on the server such as when an employee has been promoted, PUT would update his job title to the correct one,

- DELETE is used to delete one or more entries such as when an employee has quit, and should therefore be deleted from the server[10][11].

2.4.6 JSON

JavaScript Object Notation (JSON) is a data format written in plain text it is used to transmit data between server and client. JSON can represent four data types: strings, numbers, Booleans and null as well as two structure types: objects and arrays. [12]

2.5 Database

A database is a collection of interrelated data made of schemas, tables and queries. The purpose of a database is to make it easier to store and access data, which is done via a database management system.

2.5.1 Database management system

A database management system (DBMS) is a collection of interrelated data made of schemas, tables and queries. It gives users an abstract view of the data stored and allows users to easily modify and store data in the database. Databases provide important attributes such as data consistency, data isolation, data integrity, data atomic and concurrent access. It also enables users and applications to access the database at the same time.[13][14]

2.5.2 MySQL

MySQL is a relational database management system using a collection of tables to represent both data and relationships between the data. It uses SQL to insert, modify and get data from the relational database.
A table may contain multiple unique columns which stores information. The columns stores a record of a particular data type, which could be an integer, string or a char. [13][14]

### 2.5.3 Structured Query Language

Structured Query Language (SQL) is non-procedural which means that it does not require programming logic for a result. A query takes an input and returns a result based on the input. A query may contain information from more than one table and the input may contain more than one argument.

\[
\text{Select orderId from order_table where orderId=5}
\]

Is an example of how to query a column orderId from the table order_table; it returns the orderId which is equal to 5.

SQL allow users to define tables, integrity and constraints with specific data types using its Data-Definition Language. Operations that violate integrity constraints are not allowed.

A data manipulation language (DML) allows querying the database and insert, modify and delete from the database. [13][14]

### 2.5.4 Entity Relationship Model

In the entity relationship data model the collection of objects is called entities; between some of these entities there could be a relationship. An entity is a set of attributes where the attributes describes the entity. An entity may have a unique id attribute called primary key because some entities could have the same attributes, but by having a unique id the user has a way of knowing which entity is currently being worked with. An entity may include another entity’s primary key in its attributes, this is called a foreign key.

Relationship is defined as an association between several entities. For example, an order has a relationship with the order_queue.

### 2.5.5 Queries

SQL queries are made to be easy for humans to use, but the system must translate a given query into its internal form. This is similar to the work performed by the parser of a compiler. It is the system responsibility to optimize the query, which means it will minimize the cost of the query. Only a rough execution cost of a query is possible, since queries depend on many parameters such as memory available, making it difficult to compute the exact cost of the query. The cost of a query is measured in number of different resources, CPU time to execute a query and disk access, since accessing data on the hard drive is much
Parallelism within queue application
Ludvig Åberg
2017-06-02

slower than accessing data in the memory; this has a huge impact on the execution cost. The execution cost of linear search, which is a table without index and B-tree index, can be described below.

- Linear search has a cost of $t_s + b_r \times t_T$, where $t_s$ is the disk seek time to find the first block of the file, $b_r$ is the number of blocks in the file and $t_T$ is the time it takes to transfer the blocks.

- B-tree index has a cost of $(h_i + 1) \times (t_T + t_s)$, where $h_i$ is the height of the index, $t_T$ is the time it takes to transfer the blocks and $t_s$ is the disk seek time. [13][14]

2.5.6 Index

An index is used in the database to be able to find information in the database faster. There are two different kind of indices: ordered indices which are based on sorted ordering of the values, and Hash indices which are based on a uniform distribution of values across a range of buckets. The indices are best suited for different types of database applications and must be evaluated based on certain factors:

- Access type: how you access the information in the table, which can include finding information with a specific attribute or find attributes in a specific range.

- The access time for the chosen access type to find the required information.

- The time required to insert new data: this includes the time it required find the right place for the insertion of the new data and the time to update the index structure.

- The time required to delete information, which also includes the time required to find the correct information and the time to update the index structure.

- The storage space required for the index structure. [13][14]

B-tree is an index structure which takes the form of a balanced tree in which every path from the root to the leaves of the tree are of the same length. Each node in the tree has between $n/2$ to $n$ children, where $n$ is fixed for a particular tree. The B-tree has performance reduction in insertions and deletions and requires increased storage space. [13][14]

When querying the B-tree, the path is never longer than $\log[n/2] (N)$, where $N$ is the number of records and $n$ is the search key size. Since the nodes in a B-tree
There are updates on the B-tree when there is an insert or deletion in the database, and the index be updated accordingly. When an insertion occurs it could become necessary to split a node into two smaller nodes, if it has become too large. This also applies to deletion: if a node becomes too small after a deletion, it could become necessary to combine nodes. After a split or combination, it must be ensured that the balance of the tree is maintained.

The complexity of the B-tree updates require relatively few I/O operations which is important since I/O operations are expensive. The worst case for an insertion in terms of I/O operations is proportional to \( \log\frac{n}{2} (N) \), where \( n \) is the maximum number of pointers in a node, and \( N \) is the number of records being indexed. The worst case for deletion is also proportional to \( \log\frac{n}{2} (N) \), if there are no duplicated values for the search key, or if there are multiple values for the search key, the deletion process have to search in multiple records to find the correct key to delete. To ensure that there are no multiple values with the same value, the value can be made unique. The I/O operation cost in the B-tree is considered low and the speed of the operations makes B-tree a frequently used index structure.

### 2.6 Order Engine

Order Engine is a queue implementation made by Dewire to handle transactions. It is a Java EE application and using the Spring framework. It is event driven and an asynchronous application which accepts transactions from its RESTful API interface. Each transaction is sent as a JSON object to order engine. It treats each order as a transaction which means all orders must be atomic, consistent, isolated and durable.

When an order is sent to the Order Engine it is in the form of an OrderEntity which is mapped to the database table oe_order. After the OrderEntity has been persisted to the database, its order id and queue id is queued in the order queue table in the database waiting for selection and execution. The order queue table uses optimistic locking.
The scheduler in the order engine is responsible for finding executable orders from the order queue. The scheduler has an id pool where it stores the orders which can be executed. The id pool is refilled with new executable orders every second, or when the id pool is empty. The scheduler selects orders based on a first in first out per queue implementation which means that the scheduler selects the orders with the lowest order id in each queue.

A maximum of 10 orders can be executed at the same time in the Order Engine because of the thread limitation of 10. However, if the number of queues are below 10, the maximum of orders executed at the same time is limited to the number of queues. After an order has been executed it is either marked as finished or failed depending on the result from the execution. The order is then removed from the order queue and can no longer be scheduled.

The Order Engine has a web interface, giving the user an overview of the queue. The web interface shows the number of queues and the number of orders currently in the system. It is also possible to see the number of failed orders.

2.7 Related work
Dependency transactions

This paper is about dependencies within sub-transactions. It shows how dependencies can be analyzed to ensure correct execution and describes definitions, theorems and proof regarding transaction dependencies. It also describes two different ways to handle dependencies: event ordering and event enforcement. Event ordering is described as an event in a sub-transaction Ti, must precede the execution of an event in sub-transaction Tj. Commit, termination, begin, serial, begin-on-commit, and begin-on-abort are classed as event ordering dependencies.

Event enforcement is described as the execution of event in sub-transaction Ti, which requires the execution of an event in sub-transaction Tj. Strong commit, abort, weak abort, exclusion, force-commit-on-abort, force-begin-on-commit, force-begin-on-begin and force-begin-on-terminate are classified as event enforcement dependencies. Event enforcement may impose an execution order on sub-transactions.

The paper summarizes how dependencies in advanced transaction models can be analyzed to ensure correct execution.

[16]

Advanced Transactions in Enterprise JavaBeans

This paper proposes an extension to the Enterprise Java Bean (EJB) to address transaction concurrency. This because EJB suffers from weaknesses in existing transactional systems, according to the
Parallelism within queue application
Ludvig Åberg
authors. For example, EJB only allows concurrency in completely isolated cases, i.e., a flat model [17]. Transactions cannot cooperate on underlying database level or on the bean object level. This because many databases do not support sharing resources, or defining transaction-specific exceptions from the operation conflict table. In EJB beans cannot be shared between transactions unless if they are declared as stateless session beans with no identity which means that they are transaction-unaware. This is because in EJB a lock is acquired when a read only or write only is requested on a resource, the lock is released when the transaction commit or aborts. The paper proposes that by allowing a transaction to invoke read-only methods on locked bean objects the application effectiveness could be increased. A transaction cannot change its behaviour based on the state of another transaction. For example, it is not possible to mark a transaction abort-dependent on another transaction, which means it is not possible to abort a transaction depending on whether another transaction aborts.

The extension called Bourgogne Transactions addresses some of the weaknesses the author describes. This adds basic transaction primitives that represent significant begin, commit and abort transaction events. They also add advanced transaction primitives which include:

Dependences: enables transactions to establish dependencies to other transactions, such as an abort dependency.

Resource sharing; a transaction can give another transaction permission to access its data.

Delegation; a transaction can move data objects associated with it to another transaction. The transaction that accepts the data becomes responsible for the commit and abort of the operations.

However, the extension does not introduce a concept for asynchronous or queued transactions. [18]
3 Methodology

This chapter presents the methodology of the thesis: it will describe how the work and experiment have been conducted.

3.1 Research

Research will be conducted to find out how to enable the queue structure to execute tasks in parallel when it comes to dependencies, while at the same time supporting transaction properties. The research will be based on scientific papers, books and online sources.

3.2 Performance evaluation

To evaluate the implemented queue structure, performance tests will be conducted. The performance tests will measure the queue structures performance in regards to load and throughput. The measured values will be presented in Chapter 5.

The evaluation will consist of filling the implementation with a certain number of orders and queues with different types of properties to test different possible scenarios. The different scenarios that will be tested will have all of the orders on the same level, which means that all of the orders in the queue are executable and where only one order per queue is executable at a time. These scenarios will be tested with 50,000 orders generated, distributed evenly between the queues. Each test is run twice, and after each test the database is cleared to make sure that each test is run with an empty database.

The measurements will be made on time of insertion in the database tables oe_order and oe_order_queue, how many orders the scheduler fetches and how much time each fetch requires, how many orders completes per second, and how long it takes for the endpoint to receive a response. The measurements were conducted on a MSI GE620DX laptop with an Intel Core i5-2430M processor and 8 GB of ram.

3.3 Implementation

The implementations of the modified application and the test server have been done using Java EE in an IDE called Eclipse. The frameworks Spring, RESTEasy, JPA, Angular and D3 have been used for the order engine implementation. RESTEasy was used to enable communication to the server, and Angular and D3 were used to visualize the queue tree.
4 Implementation

This chapter presents the queue structure proposals and describe the benefits and cons of each queue structure proposal. The proposed modifications will be presented and described.

4.1 Queue structure proposal 1

Figure 2: Overview queue structure 1

In queue structure 1, each node knows all descending children, and the distance to each of them. This means that one row is required for each child of a node. The theoretical size of the table will be \( O(n^2) \) in the database, where \( n \) is the number of unique nodes. It benefits from fast subtree lookups in the structure, but suffers from costly insertions and deletions because of its size.
4.1.1 Insertion

Inserting a node in the queue structure can be done in two ways: one is to insert it last or to insert it in a specific position. To insert a node last in the queue, and in first in first out order, the insert will query the root node for the maximum length. It will then choose the node with the highest orderId as the parent of the inserted node. Next, it will search the tree and insert the path of the new node. The execution cost for an insert with an index is high due to the size of the structure, which increases the height of the index, as it requires more data transfers. Without indexes, the execution cost is also high due to the data transfers, in addition, it suffers from searching the table which is $O(n)$ complexity. The insertion will cause writes to $n$ rows in the database, where $n$ is the number of unique nodes: this is because each parent node knows all its children.
4.1.2 Deletion

To delete a node in the queue structure, it searches the tree and deletes each entry that has a connection to the deleted node. This is because each node knows the distance to each descendant child nodes. The execution cost for a deletion is high for both with and without index because of the amount data that needs to be deleted. The deletion of one node will cause a write to n rows in the database, where n is the size of the queue.
After the deletion, the queue structure will no longer include nodes 1, 2, 3 and 4 and all children connections will be deleted. Nodes 5, 6 and 7 will be next in line for execution.
Finding subtrees is done by querying the children of a node. This returns all the children and the distance each child have to the start node, which shows the subtree of the chosen node. The complexity of finding a subtree is $O(\log n)$ with index, and without $O(n)$. 

Figure 6: Example of finding subtrees in queue structure 1
4.2 Queue structure proposal 2

Tree structure proposal number 2 is an adjacency list [21] and is a simplified model. The main goal of proposal 2 is to reduce the size of the tree structure in the memory, and thus reduce read and write operations. The tree is structured so that each of the nodes only knows its parent, and there is no root node without value. By having only one row for each order the size of the graph will only be $O(n)$, and inserting and deleting orders from the tree will only involve writing to one row in the database.
4.2.1 Insertion

Inserting node 11 in the tree will only be a write operation, and it is inserted on the same level as other orders that allow parallel execution. A requirement is to examine what kind of order 10 is, and if 11 can be run in parallel to order 10 then order 11 will examine order 10 siblings, and if order 11 can be run in parallel. If order 11 and the siblings of order 10 can be run in parallel, order 11 uses the same parentId as order 10, and is thus on the same level. The execution cost of inserts is lower than proposed model 1 due to the smaller table size and the smaller data transfers. The insert will only do a write in one row of the database which is the inserted value and its parent.
If order 11 cannot be run in parallel with order 10 or its siblings, then order 11 will set order 10 as its parent since it is the order with the highest id on that level. This is to ensure the first-in-and-first-out queue structure.
4.2.2 Finding nodes

Finding the children of node 8 is done by querying the nodes with parent id 8, in this case order 9 and 10.

The queue structure allows finding subtrees, but the cost with every level of the tree increases exponentially. Since it has to compare the orderId and the parentId of the deeper level for every level in the tree. The complexity of finding subtrees in the tree structure is $O(n^d)$ where $d$ is the depth of the subtree.
4.2.3 Deletion

Deletion is done by selecting an order, in this case the top nodes 1, 2, 3 and 4. The operation only involves one row per node to be deleted. Since the queue implements the first-in-first-out queue structure, the children of nodes 1, 2, 3 and 4 does not need modification. In this case, the children of node 4, node 5, 6 and 7, do not need to modify their parent; they will simply point to a node which does not exist by selecting orders with the smallest parent id. This ensures that the orders in question are next in queue for execution.

Figure 11: Example of deletion in queue structure 2
Figure 12: Example of subsequent deletion queue structure 2

Figure 12 shows what the structure will look like after deletion: nodes 5, 6 and 7 points to node 4, the dashes surrounding the node means that the node does not exist in the database.

The execution cost for the selection will be lower than for structure 1 because of the smaller table size, and less data is transferred. The deletion will only affect one row per node deleted in the database, which is the row where the deleted value is.
4.3 Parallel orders implementation

Queue structure 2 was selected because of its superior properties, such as fewer writes which means faster insertions and deletions.

To use the proposed queue structure in Order Engine, modifications are required in the implementation. Since performance is crucial for Order Engine, a requirement is that the modifications impact the performance as little as possible, but still carry out the work required to be able to handle the new queue structure.

The proposed solution adds dependencies to the orders before they enter order engine, and also adds logics to Order Engine to handle the dependencies, which is followed by putting the orders in the correct position depending on the dependencies of each order. To allow this, two new columns were added to the oe_order table which is the category defining the type of order and dependencies, which in turn defines what kind of order categories an order can run in parallel with. In addition, the OrderEntity needs to be modified to save and get the new database changes.

A new column was added to the oe_order_queue table, called parent_id. This column specifies which order must be executed prior to the current order. The OrderQueueEntity needs to be modified to be able to save and get the new database changes.

When an order arrives it is saved to the oe_order table in the database after insertion into the oe_order_queue table, but before it is saved it must find which order it should set as parent. It starts by selecting the order by using the highest order id in the same queue with this query:

"SELECT oe FROM OrderEntity oe WHERE (SELECT MAX(op.orderId) FROM OrderQueueEntity op WHERE op.queueId=?1)=oe.orderId"

It compares the dependencies between the current order and the selected order, and if it the dependencies do not match, the current order sets the selected order as parent id. However, if the dependencies match, it must evaluate the dependencies between the current order and the selected order siblings, which is done by using the query:

"SELECT oe.dependencies, oe.category FROM OrderEntity oe WHERE oe.parentId=?2 AND oe.queueId=?1 AND EXISTS(select oq.parentId FROM OrderQueueEntity oq where oq.parentId=?2 and oq.queueId=?1) group by oe.dependencies,oe.category"

The query selects the dependency and category column from the oe_order table and uses the same queue id as the current order but uses the selected orders parent id to find the children. It executes a group by on the dependency and category column. This is done to prevent fetching too many orders, which could result in a huge performance drop. Instead, this query simply fetches each
unique dependency and category combination, and if one does not match the dependency comparison, the selected order with maximum order id is selected as parent. If it matches all the combinations, it selects the parent of the order with the maximum order id as parent.

The pseudo code for finding the parent order id:

```
Fetch maximum order from queue
if current order !contains dependencies of maximum order id
    return maximum order id as parent id
fetch maximum orders siblings unique dependencies
for every unique dependency
    if current order !contains dependencies of current sibling
        return maximum order as parent id
if current order contains dependencies of siblings and max order
    return max order parent id as parent id
```

The scheduler has been modified to select executable orders based on their parent id and queue id. It selects orders with the lowest parent id from each queue using the query:

```
SELECT order_id FROM(SELECT MIN(parent_id) minpid, queue_id qid
FROM oe_order_queue GROUP BY queue_id) t INNER JOIN oe_order_queue
ON t.minpid=oe_order_queue.parent_id AND t.qid=oe_order_queue.queue_id
```

Because of the limitation in JPL, this query is done as a native query instead of a JPL query.
After modification, the queue and scheduler in the modified version looks as shown in figure 13. The queues allow parallel orders within a queue as seen in queue A. The scheduler fetches every order on the highest level in each queue and puts them in the id pool then the threads selects a random order from the id pool and executes it.

### 4.4 Visualization

A new page was created on the order engine implementation; the function of this page is to visualize the queue structure and execution of the orders. It was created with the angular.js and d3.js framework. The angular controller is responsible for fetching the orders in the queue and sending them to the visualization angular directive. The visualization directive builds up an internal tree structure of the orders.
The figure shows the implementation of the visualization. Each circle is an order and the text next to it is the id of the order. The lines point toward the parent of an order. The heritage starts from the top and moves downwards in the tree, the highest orders in the tree are the parents to the second highest orders and so on. A yellow circle indicates waiting for execution, and when the circle turns to orange it indicates execution. Regardless of whether the execution succeeds or fails the circle will be removed from the tree and the orders next in line will be executed.

4.5 Test server implementation

A test server was implemented to test and verify the result on the Order engine implementation. It generates orders with certain properties depending on what is going to be tested. It can generate a user specified number of parallel orders and queues with an even distribution of orders in each queue, parallel orders are defined as all orders are executable.

The server implementation can generate a user specified number of sequential orders and queues with an even distribution of orders in each queue. Sequential orders means that only 1 order per queue can be executed at a time.

The test server creates an internal tree structure for the generated orders, which is in order of expected execution to verify that the implemented queue structure is correct.

The test server uses a REST interface to accept http requests, which in turn are used used to communicate with the test server. The REST urls are following:

- GET localhost/Testserver/controls/success; this returns the amount of succeeded orders processed by the server.
- GET localhost/Testserver/controls/controlorders; this request calls on the internal method controlOrders, which controls all generated orders
and returns information about whether the orders have been executed correctly or incorrectly.

- PUT localhost/Testserver/controls/parallelorders/{amount}/{queuesamount}; this request accepts a user-defined number and queues amount which are used to generate a user-defined numbers of parallel orders and queues.

- PUT localhost/Testserver/controls/sequentialorders/{amount}/{queues-amount}; this request accepts a user-defined number and queues amount which are used to generate a user-defined number of sequential orders and queues.

The implementation has 6 Java classes: Order, Order Generator, Order Handler, Order Checker, Send Order and CORSFilter. The order class is a class that contains all necessary information about the order such as: dependencies, queue, category, id, parent, status, operation, and children.

The order generator class is responsible for generating orders with queue id, dependencies, categories and id. It generates unique random queues up to the number specified by the user. The text length of the queue is limited to 6 letters to avoid performance loss due to a too long queue text. It then assigns the generated orders to the queues evenly. When an order is generated it is inserted into an internal tree structure which is used to control the received orders in a later phase.

The Order handler class uses the JAX RS framework and is responsible for responding to the order engine application, either with response status 201 (created), if the order received has been processed with success, or with response status 500 (internal server error), if the order received has been processed with failure. [19]

The Order checker class is responsible for controlling if the received order has been transmitted in the correct order. It uses the internal tree structure to control the received orders. The internal tree structure is created as a matrix where each outer index is created for each unique queue, and the inner index for each order that belongs to a specific queue.

When an order is received it is sent to the order checker which enters the outer index of the tree structure based on the orders queue. It then iterates through the tree checking that the order parent and the parent’s siblings have been marked as executed. If they are marked as executed the received order have been received in the correct order and is marked as executed and tells the Order handler that the order has been processed with success. If the parent or the parent’s sibling are not marked as executed, it means that it has not been processed yet and is considered received in the wrong order. The order will be marked as
failed and the order checker will inform the order handler that the order processed failed.

The Send Order class receives the sending methods for the generated orders to a specific URL. These methods use the RESTEasy framework. In this case, the http request and address used are POST to the url localhost:8080/order-engine-api/orders, this is transmitted to the order engine REST interface to create an order in order engine.

The Cross-Origin Resource Sharing (CORS) filter was created to allow the test server to make http requests to another domain. It allows it by adding headers and http methods to the REST interface. The headers specified in the CORS filter class is:

- Access-Control-Allow-Origin: this header is given the value ‘*’, which means it will accept access from any domain
- Access-Control-Allow-Methods: this header is given the values ‘GET’, ‘POST’, ‘PUT’, ‘DELETE’ to allow these http methods on the test server
5 Results

This chapter presents the results of the performance measurement of the modified order engine and the original order engine. The measurements are done on 50,000 orders and various numbers of queues. The graphs presented are execution time per queue, orders per second, order insertion time, queue insertion time, response time, and scheduler fetch time.

5.1 Execution time per queue

Figure 15: Execution time per queue

Figure 15 shows the execution time for the modified order engine and the original order engine. Blue is for parallel orders, which is defined as orders that can be run in parallel with other orders in the same queue. Green is sequential orders that is run in the modified order engine and is defined as orders that cannot be run in parallel with other orders in the same queue. Yellow is the original order engine which only supports sequential orders. The figure shows that sequential orders for both the modified and original order engine improves for each queue that is added in the system. The parallel orders are not affected to any greater extent by the number of queues in the system.
Table 1 shows the exact time of each order and orders to be completed. The parallel orders perform better than the sequential and original orders on almost all numbers of queues. The performance gap between the parallel orders and the sequential and original orders are greatly reduced for each new queue added.

For one queue, the parallel orders perform 4.71 times faster than the original order engine, and 5.68 times faster than the modified order engine with sequential orders. For seven queues, parallel orders perform 1.58 times faster than the original order engine, and 1.48 times faster than the modified order engine with sequential orders. For 13 queues, parallel orders perform 1.065 times faster than the original implementation and 1.1 times faster than the modified order engine with sequential orders. For 14 queues, the parallel orders perform 1.13 times worse than the original order engine, and 1.01 worse than the modified order engine with sequential orders.
5.2 Orders per second

This chapter presents how many orders per second the modified and the original implementation could perform. It is shown in four graphs which represent orders per second for 1 queue, 7 queues, 13 queues, and 14 queues.

Figure 16: Orders per second, 1 queue

Figure 17: Orders per second, 7 queues

Figure 16 and 17 shows orders per second for 1 queue and 7 queues. At one queue, the parallel orders perform better than sequential orders and the original order engine, which perform the same. At 7 queues, the performance gap between parallel orders and the sequential and original order engine have closed, but parallel orders are still performing better, while sequential and the original order engine are performing equally.
Figure 18 and 19 shows the orders per second for 13 queues and 14 queues. Compared to Figures 16 and 17, the performance gap between the three different types of orders have closed even more, and at 13 queues the parallel orders perform slightly better than the sequential and original order engine. At 14 queues, however, the parallel orders perform slightly worse than the original order engine.

<table>
<thead>
<tr>
<th>Queues</th>
<th>Min [ms]</th>
<th>Mean [ms]</th>
<th>Max [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10,53</td>
<td>14,32</td>
<td>17,5</td>
</tr>
<tr>
<td>7</td>
<td>8,52</td>
<td>15,97</td>
<td>19,42</td>
</tr>
<tr>
<td>13</td>
<td>11,93</td>
<td>13,27</td>
<td>14,2</td>
</tr>
<tr>
<td>14</td>
<td>9,55</td>
<td>11,85</td>
<td>13,71</td>
</tr>
</tbody>
</table>

Table 2: Orders per second, parallel orders

<table>
<thead>
<tr>
<th>Queues</th>
<th>Min [ms]</th>
<th>Mean [ms]</th>
<th>Max [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1,81</td>
<td>2,94</td>
<td>4,1</td>
</tr>
<tr>
<td>7</td>
<td>7,25</td>
<td>10,98</td>
<td>14,98</td>
</tr>
<tr>
<td>13</td>
<td>10,2</td>
<td>11,98</td>
<td>13,49</td>
</tr>
<tr>
<td>14</td>
<td>9,42</td>
<td>11,88</td>
<td>13,8</td>
</tr>
</tbody>
</table>

Table 3: Orders per second, sequential orders
5.3 Insert order

This chapter presents the insertion time of an order to the database. Time is measured from the point when the order engine receives the order until the order has been created and saved to the database. The four graphs represents 1 queue, 7 queues, 13 queues and 14 queues respectively and are presented with insertion time over orders.

Figure 20 and 21 shows order insertion time for 1 queue and 7 queues respectively. The graphs show that the performance of all the three measured orders are performing roughly the same with a few peaks.
Figures 22 and 23 show 13 queues and 14 queues and just like the previous figures 20 and 21, the measured orders perform roughly the same with a few peaks.

<table>
<thead>
<tr>
<th>Queues</th>
<th>Min [ms]</th>
<th>Mean [ms]</th>
<th>Max [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>6,95</td>
<td>11,37</td>
</tr>
<tr>
<td>7</td>
<td>6,3</td>
<td>8,13</td>
<td>13,52</td>
</tr>
<tr>
<td>13</td>
<td>6,63</td>
<td>7,13</td>
<td>9,1</td>
</tr>
<tr>
<td>14</td>
<td>6,98</td>
<td>7,77</td>
<td>11,4</td>
</tr>
</tbody>
</table>

Table 5: Order insertion time, parallel orders

<table>
<thead>
<tr>
<th>Queues</th>
<th>Min [ms]</th>
<th>Mean [ms]</th>
<th>Max [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7</td>
<td>8</td>
<td>14,27</td>
</tr>
<tr>
<td>7</td>
<td>7,26</td>
<td>8,56</td>
<td>15,86</td>
</tr>
<tr>
<td>13</td>
<td>7,06</td>
<td>8,28</td>
<td>11,62</td>
</tr>
<tr>
<td>14</td>
<td>7,02</td>
<td>7,95</td>
<td>12,53</td>
</tr>
</tbody>
</table>

Table 6: Order insertion time, sequential orders
Table 7: Order insertion time, original orders

Table 5, 6 and 7 shows the min, mean and max value for parallel orders, sequential orders and original orders for 1 queue, 7 queues, 13 queues and 14 queues. The number of queues seems to have no impact on the insertion performance since the values do not differ much. The performance of the measured orders do not differ much, which could mean that the different types of orders do not impact the order insertion time.
This chapter presents the queue insertion time i.e., how long time is required for a created order to be inserted into the order queue database. The lower the number the better.

Figure 24 and 25 show a graph over queue insertion time for 1 queue and 7 queues, respectively. At 1 queue, the parallel orders and sequential orders are increasing linearly. The performance of parallel orders is worse than the sequential orders and both parallel and sequential orders are worse than the original orders, which are barely visible in Figure 24. Figure 25 shows that the performance gap has been smaller, but original orders are still performing better than the parallel and sequential orders.
Figure 26 and 27 show the queue insertion time for 13 and 14 queues, respectively. Compared to Figure 24 and 25, the insertion time has been reduced for parallel and sequential orders, while there is no difference for original orders.

<table>
<thead>
<tr>
<th>Queues</th>
<th>Min [ms]</th>
<th>Mean [ms]</th>
<th>Max [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12,24</td>
<td>110,9</td>
<td>281</td>
</tr>
<tr>
<td>7</td>
<td>12,25</td>
<td>27,87</td>
<td>77,17</td>
</tr>
<tr>
<td>13</td>
<td>10,27</td>
<td>18,73</td>
<td>27,26</td>
</tr>
<tr>
<td>14</td>
<td>9,82</td>
<td>19,55</td>
<td>29,38</td>
</tr>
</tbody>
</table>

Table 8: Queue insertion time, parallel orders

<table>
<thead>
<tr>
<th>Queues</th>
<th>Min [ms]</th>
<th>Mean [ms]</th>
<th>Max [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>14,2</td>
<td>35,83</td>
<td>60,26</td>
</tr>
<tr>
<td>7</td>
<td>9,84</td>
<td>14,91</td>
<td>24,46</td>
</tr>
<tr>
<td>13</td>
<td>8,81</td>
<td>12,17</td>
<td>17,79</td>
</tr>
<tr>
<td>14</td>
<td>8,31</td>
<td>11,64</td>
<td>16,1</td>
</tr>
</tbody>
</table>

Table 9: Queue insertion time, sequential orders
Table 8, 9 and 10 show the min, mean and max values of Figures 24, 25, 26 and 27. Table 8 and 9 shows that the parallel and sequential orders are beneficial for performance the more queues there are. The parallel orders mean insertion time for 1 queue compared to 14 queues is 5.67 times slower. The sequential orders mean insertion time for 1 queue compared to 14 queues is 3.08 times slower. The original orders have a constant insertion time regardless of number of queues.

![Mean queue insertion time](image)

Figure 28: Summarized Queue insertion time

Table 10: Queue insertion time, original orders

<table>
<thead>
<tr>
<th>Queues</th>
<th>Min [ms]</th>
<th>Mean [ms]</th>
<th>Max [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.72</td>
<td>1.83</td>
<td>2.23</td>
</tr>
<tr>
<td>7</td>
<td>1.67</td>
<td>1.89</td>
<td>2.28</td>
</tr>
<tr>
<td>13</td>
<td>1.81</td>
<td>1.97</td>
<td>2.51</td>
</tr>
<tr>
<td>14</td>
<td>1.71</td>
<td>1.86</td>
<td>2.3</td>
</tr>
</tbody>
</table>

Figure 28 show a summarized graph of each queue and can be seen that with more queues present the faster queue insertion time parallel and sequential orders has.
This chapter presents the response time from the point when an order is sent to the order engine until it has been created, inserted into the queue and a reply has been sent.

Figure 29 and 30 show the response time from the point when an order is sent to the order engine; the original order engine responds faster than the modified order engine in both figures. Figure 30 shows that the response time has been greatly reduced for the modified order engine.

Figure 31 and 32 show that the response time has further improved for the modified order engine, but is still behind the original order engine.

<table>
<thead>
<tr>
<th>Queues</th>
<th>Min [ms]</th>
<th>Mean [ms]</th>
<th>Max [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 11: Response time, parallel orders

<table>
<thead>
<tr>
<th>Queues</th>
<th>Min [ms]</th>
<th>Mean [ms]</th>
<th>Max [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>37,75</td>
<td>56,89</td>
<td>94,53</td>
</tr>
<tr>
<td>7</td>
<td>31,46</td>
<td>37</td>
<td>64,07</td>
</tr>
<tr>
<td>13</td>
<td>27,9</td>
<td>32,9</td>
<td>42,6</td>
</tr>
<tr>
<td>14</td>
<td>27,48</td>
<td>31,74</td>
<td>44,97</td>
</tr>
</tbody>
</table>

Table 12: Response time, sequential orders

<table>
<thead>
<tr>
<th>Queues</th>
<th>Min [ms]</th>
<th>Mean [ms]</th>
<th>Max [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>23,56</td>
<td>24,7</td>
<td>32,1</td>
</tr>
<tr>
<td>7</td>
<td>23,83</td>
<td>25,61</td>
<td>31,26</td>
</tr>
<tr>
<td>13</td>
<td>24,76</td>
<td>26,45</td>
<td>35,13</td>
</tr>
<tr>
<td>14</td>
<td>23,82</td>
<td>25,16</td>
<td>32,08</td>
</tr>
</tbody>
</table>

Table 13: Response time, original orders

Tables 11, 12 and 13 show the min, mean and max value of parallel orders, sequential orders and original orders. The mean value of the parallel orders and sequential orders are decreasing with the number of queues. Parallel orders have 3.3 times faster response time at 14 queues compared to 1 queue. Sequential orders have 1.79 faster response time at 14 queues compared to 1 queue. The original orders, however, have a constant of about 25 ms for all the queues.

### 5.6 Scheduler fetch time

This chapter presents the performance of the scheduler in the three test cases. The graphs shows how long time in milliseconds, is required for the scheduler to fetch executable orders from the queue. The figures have a log scale on the x axis.
Figure 33 and 34 shows the scheduler fetch time for 1 queue and 7 queues. It should be noted that when sequential orders and original orders are fetched, the scheduler fetches one order per queue. In Figure 33 and 34, the scheduler fetches 1 order when there are only 1 queue present and 7 orders when there is 7 queues present. While parallel orders are present, the scheduler will fetch the numbers of orders that can be executed in parallel regardless of how many queues. In this case 50,000 orders are fetched at the start. Figure 33 and 34 show that the scheduler carries out fewer fetches when parallel orders are present but the fetch time is greater. The fetch time is reduced the more fetches it carries out. This is because the executed orders are removed from the queue, and thus there are fewer orders to fetch. The sequential orders behaves the same way as the original orders, but with a greater fetch time in both cases.
Parallelism within queue application
Ludvig Åberg

2017-06-02

Figure 35 and 36 show the fetch time for 13 queues and 14 queues, respectively. The graphs show the same properties as the previously figures, but sequential and original orders carry out fewer fetches since there are more queues to fetch from.

<table>
<thead>
<tr>
<th>Queues</th>
<th>Min [ms]</th>
<th>Mean [ms]</th>
<th>Max [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3,18</td>
<td>134,1</td>
<td>260</td>
</tr>
<tr>
<td>7</td>
<td>6,33</td>
<td>131,</td>
<td>342,8</td>
</tr>
<tr>
<td>13</td>
<td>4</td>
<td>141,1</td>
<td>271</td>
</tr>
<tr>
<td>14</td>
<td>3,18</td>
<td>158,4</td>
<td>372</td>
</tr>
</tbody>
</table>

Table 14: Scheduler fetch time, parallel orders
Tables 14, 15 and 16 present the min, mean and max values of Figures 33, 34, 35 and 36. The mean value of the fetch time for parallel orders seems to increase slightly when there are more queues present. The sequential orders share the same property; the mean fetch time is increased with the number of queues. The original orders have the fastest mean fetch time of all the queues. The mean fetch time is increased by 31 ms when going from 1 queue to 7 queues, but stays around 55 ms for 13 queues and 14 queues.
6 Evaluation

The results show that when the queues were below 10 the modified order engine performed better in terms of execution time than the original order engine when parallel orders were present. When sequential orders in the modified order engine were measured it performed slightly worse than the original order engine. Above 10 queues the performance gap decreased between the tested cases and performed roughly the same. The reason why the parallel orders perform much better with a smaller number of queues is because it uses all 10 threads to execute the orders, while the sequential orders are limited by the number of queues present. This results in performance increase for the smaller number of queues for the parallel orders.

The orders per second show the same pattern, while parallel orders have a high orders per second over all queues; the sequential and original orders have a lower orders per second rate when there are few queues, and increases with the number of queues. However it should be noted that parallel orders for 14 queues seems to be somewhat off: this could be due to unexpected load on the machine it was tested.

For order insertion time the three test cases performed equally well, which means that the modifications to order engine does not impact the insertion time of orders in any way.

The queue insertion time, however, is a different matter, and the modified order engine perform worse both when it comes to parallel and sequential orders. This is because of the implemented find parent method which queries the unique categories and dependencies from the orders in the same queue as the current order, and which needs a parent thus and compares the received result with the current order. For parallel orders, it will query each order inserted at 1 queue, which will be 50,000 orders in total. However, for each queue added, the fewer orders it queries, which results in a faster insertion time. For sequential orders, the queried orders are always one since no order can be run in parallel. For the original order engine there is no method for finding and inserting a parent into the queue, and is thus it performs the same in all cases.

The response time seems to primarily depend on the order insertion and queue insertion time, which is expected since the response is sent after the insertions have completed.

The measurement of scheduler fetch time shows that when fetching parallel orders it performs worse than sequential and original orders. This is due to the scheduler fetching the entire order queue since each order is able to be executed. Fetching the sequential orders performs slightly worse than the original orders, this is because the modified order engine has a more complex query to the order queue table, which increases the fetch time of the scheduler.
7 Conclusions

Queue structure number two was selected for implementation and testing. While queue structure number one would have faster relation lookups, such as finding subtrees, children, parents and ancestors, it suffers greatly in terms of deletion and insertion due to the size of the queue structure. This kind of structure would be more suited in a more static environment where insertions and deletions are less frequent than in a queue. Queue structure number two was a simplified structure which only added a parent id to the current queue structure, but doing so added the possibility of seeing relationships between the orders, and also define dependencies between the orders. This enabled order parallelism within a queue, which was not possible before. Since the performance of the queue is crucial, the proposed queue structure was a better choice because of the smaller size and theoretically faster inserts and deletions.

To integrate the proposed queue structure into the system, modifications had to be made. The database needed some modifications to be able to store the new attributes added such as category, dependencies and parent id. More advanced database queries had to be made, such as finding parent id and selecting the correct orders, which was time-consuming because of the JPQL language, which is more limited than MySQL. It was important to have fast and effective queries to limit the performance reduction as far as possible. Due to the limitations of the JPQL, one of the queries was made as a native query, since the author could not make a fast and effective enough query in JPQL. However the native query does not take fairness into consideration in a scenario where one queue has significant more parallel orders ready to be executed than other queues the other queues may get little execution time. The author attempted to solve this by making a fair query but due to the complexity of the query the execution cost was too high.

The visualization showed how the order engine worked on the orders in a tree structure in real time. It mainly serves as a demonstration since in reality the order engine executes orders too fast, which makes it difficult to see what really happens. When demonstrating the visualization, an execution delay of about 2 seconds is used to get a clear view.

The test server implementation was created to be able to test the modified order engine more thoroughly and to be able to generate orders from a certain distribution instead of doing them by hand. The test server also controlled the received orders, and the controller was manually tested to ensure it was giving the correct result. The test server carried out the evaluation of the modified implementation much faster and easier, and contributed significantly to the end result.

The result showed that the proposed modification of the order engine increased the execution time with parallel orders when the number of queues was below 10. Since the number of orders executed scales with the number of active
queues up for to 10 queues in the original order engine, this was a good result. It enables the order engine to maximize the throughput, if parallel orders are present regardless of number of queues. However, it has downsides, as it performs worse in queue insertion time, especially when there is only 1 active queue, which results in response time. This is in addition to the scheduler having a greater fetch time than the original order engine.

7.1 Ethical discussion

The original order engine may handle sensitive customer information which should not be seen by unauthorized people. As all customer information is important, the order engine must ensure that it will deliver this information in a correct way to the receiver. The modified order engine must also ensure this, however, since adding the possibility to parallel the orders, it provides the users who define the orders with the responsibility to give each order the correct dependencies. If an order is given the wrong dependencies it may result in storage of incorrect customer information.

7.2 Future work

Future work could include optimizing the queries of the modified order engine. For example, creating a faster find parent method to decrease the queue insertion time, which was significantly greater than it was for the original order engine. Optimizing the scheduler query or making it more intelligent to prevent it from fetching the entire queue when there are many parallel orders in the queue is also something that could be dealt with in future work.

Attempting to move the queue from the database to the memory could also be of interest, this would reduce much of the bottleneck which is a result of the queue being in the database, which could increase the speed of the queue..
References


https://docs.oracle.com/cd/E17276_01/html/gsg_xml_txn/java/nestedtxn.html


https://www.tutorialspoint.com/jpa/jpa_introduction.htm

http://docs.oracle.com/javae6/tutorial/doc/bnbuf.html

http://www.objectdb.com/java/jpa/query/jpql/structure

https://d3js.org/

https://docs.jboss.org/resteasy/docs/3.0.9.Final/userguide/html_single/

[10] School of Information & computer sciences, “Representational State Transfer (REST)” [www]

https://www.w3.org/TR/2004/NOTE-ws-arch-20040211/#relwwwrest

[12] IETF, “JSON Data interchange format” [www]

Parallelism within queue application
Ludvig Åberg 2017-06-02


https://docs.gigaspaces.com/sbp/parallel-queue-pattern.html


[17] docs oracle, “Handling transactions with enterprise beans” [www]
https://docs.oracle.com/cd/E19229-01/819-1644/detrans.html

[18] Marek Prochazka, Advanced Transactions in Enterprise JavaBeans, Charles University , Faculty of Mathematics and physics November 2000

[19] w3, “Status code definitions” [www]
https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html


[22] Silberschatz, Galvin, Gagne, Operating system concepts 8th edition

[23] Edward A. Lee, The problems with threads