# Interference open cluster

*documentation & cookbook*

current revision: 2021.1

**io.digital**

## Contents

# Concepts & features

*i.o.cluster* also known as *interference open cluster* is a simple java framework enables you to launch a distributed database and complex event processing service within your java application, using JPA-like interface and annotations for structure mapping and data operations. This software inherits its name from the *interference project*, within which its mechanisms were developed.

*i.o.cluster* is a opensource, pure java software.

The basic unit of the *i.o.cluster* service is a *node* – it can be a standalone running service, or a service running within some java application.
Each *i.o.cluster* node has own persistent storage and can considered and used as a local database with following basic features:

- runs in the same JVM with your application
- operates with simple objects (POJOs)
- uses base JPA annotations for object mapping directly to persistent storage
- supports SQL queries with READ COMMITTED isolation level
- supports transactions
- supports complex event processing (CEP) and simple streaming SQL
- can be used as a local or distributed SQL database
- uses the simple and fast serialization
- uses persistent indices for fast access to data and increase performance of SQL joins
- allows flexible management of data in memory for stable operation of a node at any ratio of storage size / available memory, which allows, depending on the problem being solved, how to allocate all data directly in memory with a sufficient heap size, or use access to huge storages with a minimum heap size of java application

Nodes can be joined into a cluster, at the cluster level with inter-node interactions, we get the following features:

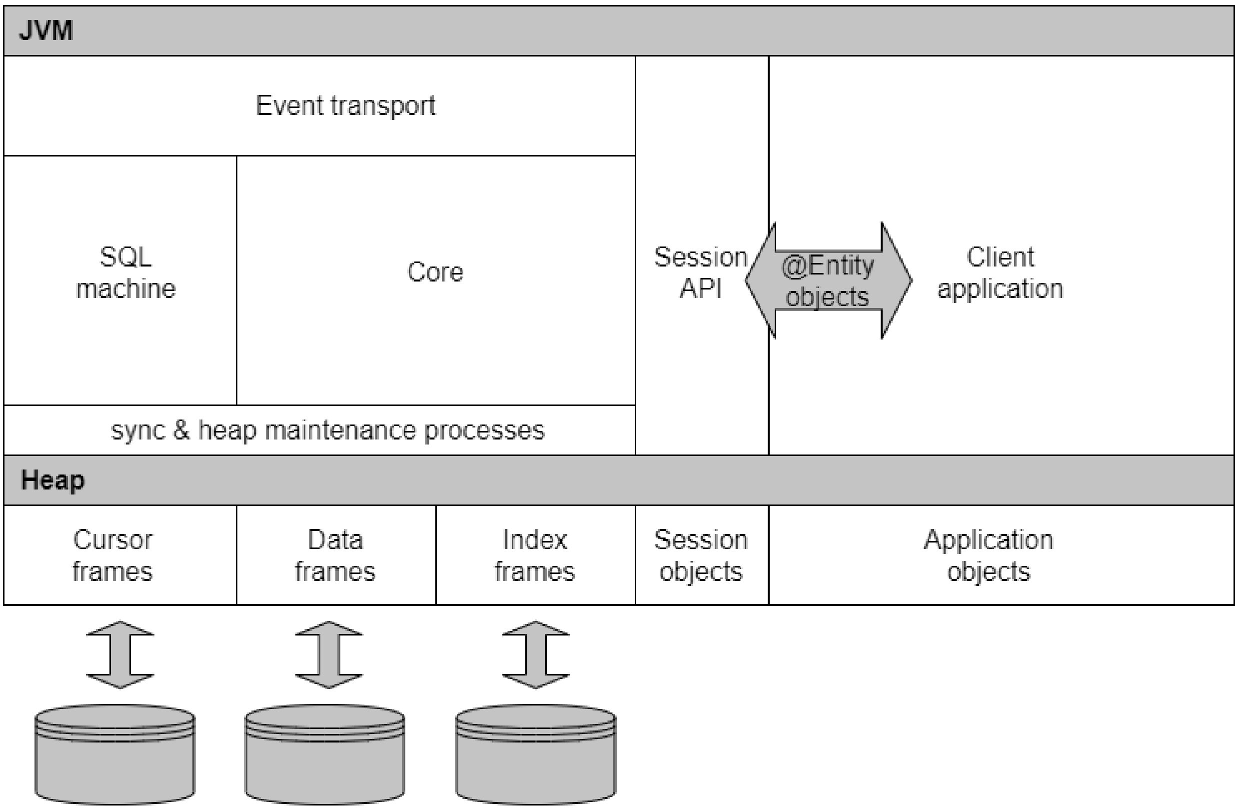
- allows you to insert data and run SQL queries from any node included in the cluster
- support of horizontal scaling SQL queries with READ COMMITTED isolation level
- support of transparent cluster-level transactions
- support of complex event processing (CEP) and simple streaming SQL
- *i.o.cluster* nodes does not require the launch of any additional coordinators

## Cluster overview

Initially, the service was designed in such a way that each node is a java application that can be launched both by sharing one JVM with the client application using the service, or autonomously. Each node uses its own storage and, being included in the cluster, replicates to other nodes all changes made on it, and also reflects changes made on other nodes.
You can start the service of each specific node inside an application and use fast access to data inside the node, as well as execute queries that will automatically scale to other nodes in the cluster. Also from your java application, you can use remote client connections to the nodes of an existing cluster without the need to deploy a full service with its own storage (see Remote Client).

Each of the nodes includes several mechanisms that ensure its operation:

- core algorithms (supports structured persistent storage, supports indices, custom serialization, heap management, local and distributed sync processes)
- SQL and CEP processor
- event transport, which is used to exchange messages between nodes, as well as between a node and a client application



## Quick Start Application

The *iocluster-test* application shows example of using the basic use cases. Before starting and using, read the following documentation.
Consider a basic example when the interference open cluster service used as a local persistent layer of the application and runs in the same JVM with the application.

To get started with *i.o.cluster*, you need to download the source code of the current release (for example 2021.1) on github, build it using maven and install the jar into your local maven repository (Minimum requirements for build: JDK 1.8, Maven 3), then add the following dependency to the pom.xml of your project:

```
<dependencies>
    <dependency>
        <groupId>su.interference</groupId>
        <artifactId>interference</artifactId>
        <version>2021.1</version>
    </dependency>
    ...
</dependencies>
```

Next, specify the necessary set of keys in the project (application) settings (jmxremote settings is optional):

```
-Dsu.interference.config=interference.properties
-verbose:gc
-Xloggc:<your_log_path>/ioclustergc.log
-XX:+PrintGCDetails
-XX:+PrintGCDateStamps
-XX:+AggressiveOpts
-Xms1G
-Xmx4G
-XX:MaxMetaspaceSize=256m
-XX:+UseStringDeduplication
-XX:ParallelGCThreads=4
-XX:ConcGCThreads=2
-Dlogback.configurationFile=config/app-log-config.xml
-Dcom.sun.management.jmxremote
-Dcom.sun.management.jmxremote.port=8888
-Dcom.sun.management.jmxremote.authenticate=false
-Dcom.sun.management.jmxremote.ssl=false
```

To run a single local interference node, you can use the standard supplied interference.properties configuration. Note that file interference.properies should be within /config subdirectory. Next, see the configuration section. Then, add following code into initializing section of your application:

```
Instance instance = Instance.getInstance();
Session session = Session.getSession();
instance.startupInstance(session);
```

where Instance is `su.inteference.core.Instance` and Session is `su.interference.persistent.Session`.

## Service as standalone

This option can be used when the cluster node is used solely for the purpose of further horizontal scaling of the data retrieving mechanism:

```
java -cp interference.jar
-Dsu.interference.config=interference.properties
-verbose:gc
-Xloggc:<your_log_path>/ioclustergc.log
-XX:+PrintGCDetails
-XX:+PrintGCDateStamps
-XX:+AggressiveOpts
-Xms1G
-Xmx4G
-XX:MaxMetaspaceSize=256m
-XX:+UseStringDeduplication
-XX:ParallelGCThreads=4
-XX:ConcGCThreads=2
-Dlogback.configurationFile=config/app-log-config.xml
-Dcom.sun.management.jmxremote
-Dcom.sun.management.jmxremote.port=8888
-Dcom.sun.management.jmxremote.authenticate=false
-Dcom.sun.management.jmxremote.ssl=false
su.interference.standalone.Start
```

## Entities

For use with the interference open cluster service, the entity class must compatible with the JPA specification, i.e. satisfy the following minimum criteria:

- be annotated with @Entity annotation
- all persisted columns should be annotated as @Column
- all other columns should be annotated as @Transient
- contain an empty public constructor
- it is mandatory to have a unique identifier field marked with @Id annotation
- class should not be final

An example of the simple annotated POJO with indexed column below:

```
@Entity
@Table(name="Dept",indexes={@Index(name="DeptPk", columnList="deptId", unique=true)})
public class Dept {

    @Column
    @Id
    private int deptId;

    @Column
    private String deptName;

    public Dept() {

    }

    public int getDeptId() {
        return deptId;
    }

    public void setDeptId(int deptId) {
        this.deptId = deptId;
    }

    public String getDeptName() {
        return deptName;
    }

    public void setDeptName(String deptName) {
        this.deptName = deptName;
    }

}
```

The current version also supports JPA annotations **`@Transient`**, **`@Table`**, **`@Column`**, **`@Index`**, **`@GeneratedValue`**. Validation of field values and unique constraint only supported for @Id-annotated fields.

Attention! The field marked with the **`@Id`** annotation must be filled with unique data. It is highly recommended that you always use the **`@GeneratedValue`** annotation in conjunction with this annotation to automatically generate unique keys.

Examples of annotated entities can be found in the *interference-test* application.

## Entity class registration

For any further operations, we need a user session:

```
Session session = Session.getSession();
```

Register method registers the Dept class in the system directory, stores metadata to the system persistent storage, then creates and loads the corresponding proxy class for it (register Dept entity class, as example):

```
session.registerClass(Dept.class);
```

This action is performed only once; then, the system will use persistent metadata when accessing this object. If you need to perform a table structure change, then this table must be completely deleted with ALL the data and registered again.

## Create instance of entity class

All further operations of inserting, changing, or retrieving data must be performed on an object obtained with the participation of the factory method newEntity(class)

```
Dept dept = (Dept) s.newEntity (Dept.class);
```

## Persistent operations

*i.o.cluster* implements the most simple data management model, which is based on several standard JPA-like methods on the Session object, which can be obtained by calling the static method Session.getSession():

- `persist()` - placing an object of @Entity-annotated class in a storage
- `find()` - find and returns an object from storage by a unique identifier
- `execute()` - execution of a SQL query and returns ResultSet object which contains a poll() method to retrieve the query execution results
- `commit()` - committing a transaction
- `rollback()` - rollback a transaction

The newly created object is not persistent, in order to save it in the database and make accessible to other sessions you need to execute the following operation:

```
session.persist(dept);
```

so, the session supports the following methods:

`session.newEntity(class)` – creating a proxy instance of a user class. I repeat once again that in order to use all the features of the Interference engine, all the operations described below must be performed with the proxy object constructed by this method after registering the user class, and not with the user class instance obtained through new. At least in this case, transactions will not be supported, and in the current version this will most likely lead to an error.

`session.persist(object)` – inserts a newly created instance into the database or saves changes to an existing one.

To change an object (update) you need to use `find()`, then make the necessary changes to the object, then execute persist. If identifiers are set up by the application, then using persist it is possible to change (update) an object in the database both in the above way (find - change - persist) and using a newly created object with the necessary identifier. The changed object is locked for changes from other sessions (transactions) until commit or rollback are executed (see below for transactions).

It should be noted that for objects with **`@NoCheck`** annotated entifier, NO checks are made for the existence of an existing object with such an id. In this case we rely entirely on the correctness of the mechanism for generating identifiers.

**`@NoCheck`** annotation was designed for use with CEP tables that do not use indexes. In the case of bulk inserts, there will be no decreases performance with lengthy checks. For mass inserts, it is recommended to use just such an approach, because it is significantly faster.

`session.delete(o)` – removes an object from the database. The object is locked for changes from other sessions (transactions) until commit or rollback (see below - transactions).

`session.purge(o)` – removes an object from the database regardless of committed transaction state. This feature should be used within CEP processing, since SELECT STREAM returns uncommitted data (supports DIRTY READS).

`session.find(class, id)` – returns an entity from the database by identifier

**Note:** in the current version, the following types are supported for the identifier field:

```
Int, long
java.lang.Integer, java.lang.Long
java.util.concurrent.AtomicInteger
java.util.concurrent.AtomicLong
java.lang.String
```

## Transactions isolation

Interference open cluster supports transactions for read / write operations. The default isolation level is *READ COMMITTED*, which means that all changes made in any transaction will only apply to those retrievable datasets whose retrieval started after commit was executed in the original transaction, regardless of the retrieval duration.

All the above methods of extracting or saving data automatically create a transaction, if it has not yet been created.

To complete the transaction and apply changes to this transaction for the remaining sessions, we perform:

```
session.commit();
```

To complete the transaction and rollback the changes, we use:

```
session.rollback();
```

In addition, the following two methods can be used:

`session.lock()` – the current transaction receives an object lock and creates an undo-snapshot for it. Returns the current object, i.e. similar to the find method, but with getting an object lock for subsequent possible changes. The object remains locked until commit or rollback.

`session.startStatement()` – start of the statement - all selections in the current session will return consistent data at the time the start command is executed in the current session until commit or rollback are executed. At the same time, it must be understood that the above methods for extracting data (`find()`, `execute()`) execute the start method automatically at each start.

As described above, and it is important to understand that the persisted entity class instance (or returned by `find()` method) is shared (interference open cluster does not create separate instances for each session) and supports isolation. So, changing the instance field in one session will not be visible in another session until commit is called in the first. It does not require re-retrieving of the Entity class instance.

## Executing SQL queries

SQL query is run using following method:

```
session.execute(query);
```

where query is a string value contains SQL query.

Execute method returns `su.interference.sql.ResultSet` object which contains set of `su.interference.proxy.GenericResult` objects which is contains result row data.

Table names are indicated as fully qualified class names, for example, `su.interference.entity.Dept`.

<u>All class and field names are case sensitive.</u>

The field names in the `ResultSet` are the specified names or aliases, if the table alias was used in the field naming, the name in the `ResultSet` will look like the table alias + field name, for example, `ddeptName` for the `d.deptName` specified in the request:

```
String sql = "select d.deptName, e.empName, e.descript from
su.interference.entity.Dept d, su.interference.entity.Emp e where
d.deptId = e.deptId";

ResultSet rs = s.execute(sql);

Object result = (GenericResult) t.poll(s);

while (result != null) {

    Object o = result.getValueByName("ddeptName");
    result = (GenericResult) t.poll(s);

}
```

All data inside the ResultSet will be consistent at the time the SQL query starts.

## SQL SELECT clauses available in 2021.1 release:

```
SELECT [GROUP_FUNC(alias1.group_column),…] alias1.column_name1,
alias2.column_name2,…

FROM fully_qualified_class_name1 alias1, fully_qualified_class_name2
alias2,…

[WHERE Condition1 AND/OR Condition2 … ]

[GROUP BY alias1.column_name1, alias2.column_name2, … ]

[ORDER BY alias1.column_name1, alias2.column_name2, … ]
```

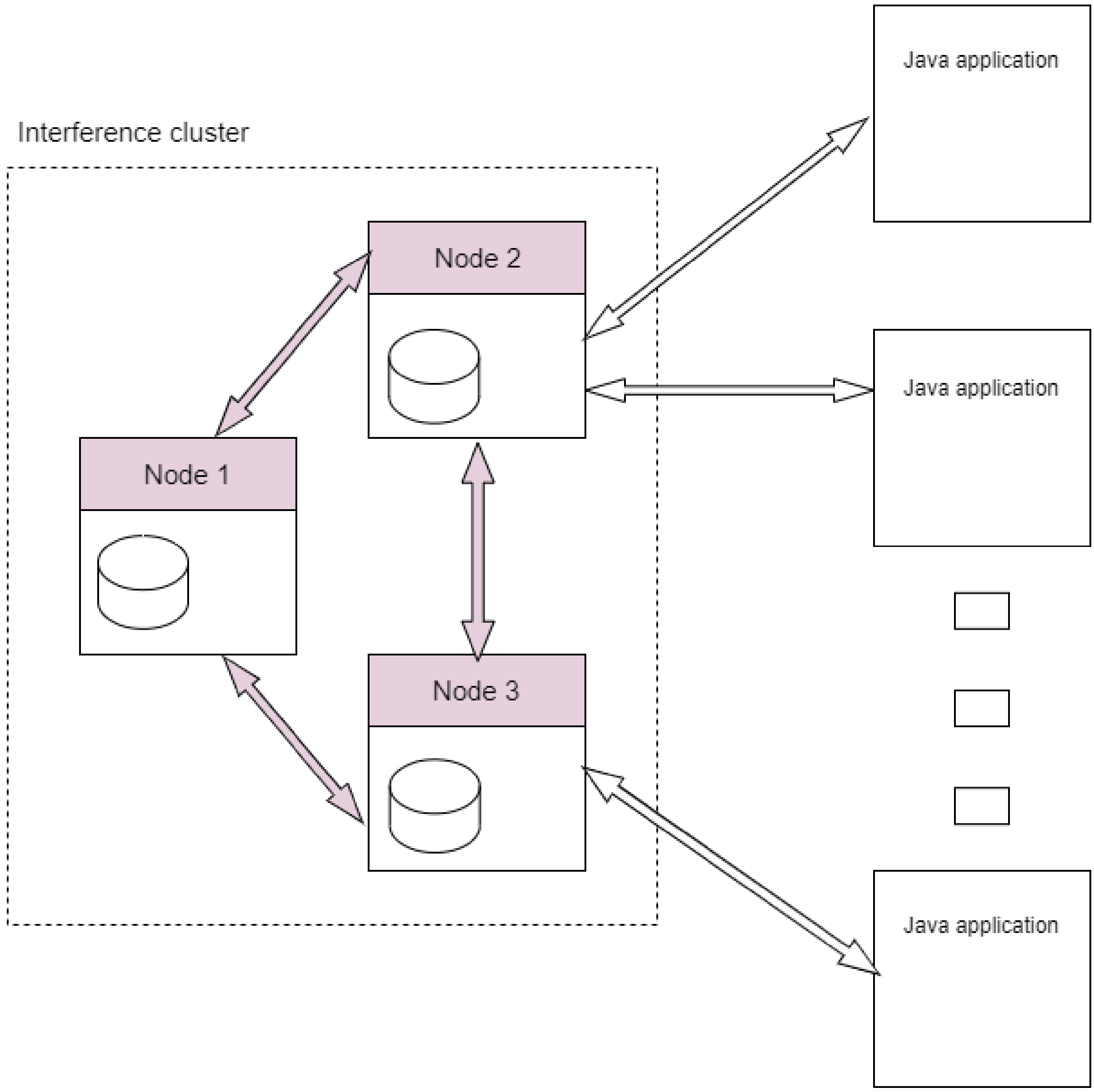
where ConditionN is standard SQL condition, e.g.:

- alias.num_column_name = 12345
- alias.string_column_name = ’string constant’
- alias.date_column_name = ’01.01.2019’ (use config.dateformat)
- alias.column_name IN / NOT IN [12345, 12346, …
- alias1.column_name1 = alias2.column_name2
- instead equals sign ( = ) may use <, >, <>, <=, >=

GROUP_FUNC is one of next group function: COUNT(), SUM(), MIN(), MAX(), AVG(). SELECT and FROM clauses is mandatory. In […] described optional clauses.

## Remote client

You can use a remote client to access interference open cluster data from another JVM. To do this, you need to create a connection to any node using the `Session.getRemoteSession()` method. The remote client uses the same transport mechanism that is used for inter-node communication within the cluster:




To create a remote session, you need to specify the host parameters for connecting to the remote cluster node, as well as the parameters of your current host for callbacks from the transport service. The client port on your client connection can be any of the available (unused by other applications):

```
String host = "remote.host.com"; //one of cluster nodes
int port = 8050; //service port
String callbackHost = "client.host.com"; //your client host
Int callbackPort = 8099; //your client port

RemoteSession session = Session.getRemoteSession(host, port,
callbackHost, callbackPort);
```

This method creates a Session object inside the cluster and returns its identifier. The remote application can perform further actions through calls to methods of the RemoteSession object using standard methods inherent in Session.

Earlier it was said that for any operations with an entity class object, this class must be registered within the interference open cluster service.
To register, we must perform the following action:

```
session.register(YourEntityClass.class);
```

If the class has already been registered before, an exception will be thrown.

Inserting or changing some object into the table:

```
session.persist(o);
```

Search for an object by identifier:

```
Object o = session.find(YourEntityClass.class, id);
```

Execution of an SQL query, and an instance of the RemoteResultSet object will be returned:

```
RemoteResultSet rs = session.execute("select c.id, c.description from your.domain.YourEntityClass c");
```

Polling an object from a result set:

```
GenericObject g = rs.poll();

while (g != null) {
    g.getValueByName("cid");
    g.getValueByName("cdescription");
    g = rs.poll();
}
```

Commit changes:

```
session.commit();
```

Rollback of changes:

```
session.rollback();
```

## Direct access to entity storage

For some cases, you can use direct access to table storage to retrieve data, as if you were working with a huge collection, without the need for run SQL query:

```
Session session = Session.getSession();
session.startStatement();

Table t = Instance.getInstance().getTableByName(Dept.class.getName());
boolean hasNext = true;

while (hasNext) {
    Dept dept = (Dept) t.poll();
    if (dept == null) {
        hasNext = false;
    }
    ...
}
```

## Distributed persistent model

To include a node in the cluster, you must specify the full list of cluster nodes (excluding this one) in the cluster.nodes configuration parameter. The minimum number of cluster nodes is 2, and the maximum is 64 (for more details, see cluster configuration rules below).

**Attention!** *Cluster.nodes parameter should be filled completely before the first start of any of the nodes, and subsequently should not be changed. Then, you can start nodes in any order and at any time*.

After such configuration, we may start all configured nodes as cluster. In this case, all nodes will be use specific messages (events) for provide inter-node data consistency and horizontal-scaling queries.

Interference open cluster is a decentralized system. This means that the cluster does not use any coordination nodes; instead, each node follows to a set of some formal rules of behavior that guarantee the integrity and availability of data within a certain interaction framework.

Within the framework of these rules, all nodes of the Interference open cluster are equivalent. This means that there is no separation in the system of master and slave nodes - changes to user tables can be made from any node, also all changes are replicated to all nodes, regardless of which node they were made on.

Running commit in a local user session automatically ensures that the changed data is visible on all nodes in the cluster.

# Distribute rules

The concept of interference open cluster is based on a simple basic requirement, which can be literally expressed as follows: we must allow insertion and modification of data at the cluster level from any node, and we must allow data retrieval from any node, using as much as possible the computing resources of the cluster as a whole. Further, we accept the condition that all cluster nodes must be healthy and powered on, if any of the nodes has been turned off for a while, it will not be turned on to receive data until her storage is synchronized with other nodes. In practice, in the absence of changes in the moment, this means that there are identical copies of the storage on the cluster nodes. To prevent conflicts of changes in cluster, several lock modes are used:

- table level (a session on a node locks the entire table)
- frame level (a session on a node locks a frame)
- disallowed changes for non-owner nodes

here it is necessary to explain in more detail: all data inserts on a certain node are performed into a frame which was allocated on the same node, for which, in turn, the node is the owner. This is done so that when there are simultaneous inserts into a table from several nodes at once, there are no conflicts during replication. Subsequently, this distinction allows us to understand whether or not to request permission to change the data in the frame at the cluster level or not. Moreover, it allows us to implement a mode when changes to frames on a non-owner node are prohibited. This mode is used on cluster nodes if one or more other nodes become unavailable (we cannot know for certain whether the node is down or there is a problem in the network connection).

So, let's repeat again:

- all cluster nodes should be equivalent
- all changes on any of the nodes are mapped to other nodes immediately
- data inserts are performed in local storage structure, and then the changes are replicated to other nodes.
- if replication is not possible (the node is unavailable or the connection is broken), a persistent change queue is created for this node
- the owner of any data frame is the node on which this frame has been allocated
- data changes in node own dataframe are performed immediately, else, performed distributed lock for dataframe on cluster level
- if cluster is failed (some node are offline or connection broken), all data changes are not allowed or changes in only node own dataframes allowed
- the cluster uses the generation of unique identifiers for entities (@DistributedId annotation) so that the identifier is unique within the cluster, but not just within the same node
- the cluster does not use any additional checks for uniqueness, requiring locks at the cluster level

## @DistributedId annotation

Using standard @Id, @GeneratedValue annotations implies the generation of unique values within a single node. If your distributed application is guaranteed that data insertion process will be performed on only one specific node, then this pair of annotations is enough. If the data can be inserted on different nodes, you must use the @DistributedId annotation with the above pair of annotations. This annotation guarantees the uniqueness of the generated identifier within the cluster and is highly recommended for use with @Id and @GeneratedValue.

## SQL horizontal-scaling queries

All SQL queries called on any of the cluster nodes will be automatically distributed among the cluster nodes for parallel processing, if such a decision is made by the node based on the analysis of the volume of tasks.

If some node becomes unavailable during the processing of a request (network fails, service stopped), the task assigned for this node will be automatically rescheduled to another available node.

## Complex event processing concepts

So, we must allow insertion and modification of data at the cluster level from any node, and we must allow data retrieval from any node, using as much as possible the computing resources of the cluster as a whole.

The next concept of interference open cluster is that any table is at the same time a queue, in particular, using the SELECT STREAM clause, we can retrieve records in exactly the same order in which they were added. Usually, at the cluster level, the session.persist() operation can be considered as publishing a persistent event. Based on our basic distribution rules, we send this event to all nodes.

Interference open cluster does not currently support the standard DML UPDATE and DELETE operations, instead for bulk table processing (including the optional WHERE clause) we have implemented PROCESS and PROCESS STREAM clauses that allow us to process each record from a selection of one of the EventProcessor interface implementations. On the one hand, this approach allows us to obtain results similar to those that we would achieve using UPDATE and DELETE, on the other hand, it significantly expands the possibilities for custom processing of records, allowing full event processing. For the sake of fairness, it is need noting that you can get similar results using standard SELECT and SELECT STREAM, using some custom code to process the result set, but PROCESS and PROCESS STREAM implement processing at the core level of the cluster, which significantly improve the performance, second, this statements are launched at the cluster level and provide a ready-made implementation for distributed event processing.

In order to create a custom EventProcessor implementation, we need to implement two methods:

boolean process(Object event)

in this method, custom event handling should be implemented, in case of successful processing, true is returned.

boolean delete()

if this method returns true, the record will be deleted from the table upon successful completion of processing (the process method returned true).

Next, we can use the following query:

PROCESS fully_qualified_class_name alias

WITHIN fully_qualified_event_processor_class_name

[WHERE condition1 AND/OR condition2 … ]

[ORDER BY alias.column_name … ]

For example, it might look like this:

```
String sql = "process su.interference.entity.SomeEvent d within su.interference.processor.SomeEventProcessor where d.eventType = 1";

ResultSet rs = s.execute(sql);
```

The PROCESS statement allows to process records from one specific table in batch mode, currently the query does not support any joins to other tables. The PROCESS statement is a distributed operation and performs processing on all nodes of the cluster, for which it locks the table at the cluster level while the query is running for any other PROCESS statements may be launched from other nodes or from other sessions.
This processing is performed inside a transaction, therefore, after execution, we need to explicitly apply commit or rollback.

## Online complex event processing

Except standard batch processing, interference open cluster supports online complex event processing using SELECT STREAM (and PROCESS STREAM for launch with EventProcessor implementations) clause in SQL statement. The basic differences between of a streaming query and the usual one are as follows:

- the `session.execute(...)` method returns a `StreamQueue` object, which is an implementation of ResulSet,
- the request is executed asynchronously until `StreamQueue.stop()` method will be called or until the application terminates,
- the `StreamQueue.poll ()` method returns all records previously inserted into the table and according to the WHERE condition (if exist) and continues to return newly added records,
- each `StreamQueue.poll ()` method always return next record after last polled position within the session, so that, provided that the SQL request is stopped and called again within same session, data retrieve was continued from the last fixed position, in another session data will be retrieve from begin of table,
- unlike usual, a streaming request does not support transactions and always returns actually inserted rows, regardless of the use of the `commit()` method in a session inserting data (DIRTY READS). However, it should be keep in mind that `rollback()` physically deletes data, so use it in this context with careful.
- The `session.closeStreamQueue` method stops the current stream query execute.

The simplest example is a query from a table that returns either all records or filtered by some condition. Such a request can be used to broadcast events from node to node or to generate alerts of a certain type:

```
String sql = "select stream e.empName, e.descript from
su.interference.test.entity.Event e where e.eventType = 1";

StreamQueue q = (StreamQueue) s.execute(sql);
while (true) {
    Object o = q.poll();
    ...
}
```

Tumbling windows

This example implements the so-called streaming aggregation and assumes that the inserted records are analyzed in a strictly defined order and for each group of such records one output record will be generated using group functions such as AVG, COUNT, MAX, MIN, SUM:

```
String sql = "select stream sum(e.eventValue), e.groupValue from
su.interference.test.entity.Event e group by e.groupValue";

StreamQueue q = (StreamQueue) s.execute(sql);
while (true) {
    Object o = q.poll();
    ...
}
```

A necessary and important note: since the insertion can be carried out in several threads, the order in which records are analyzed and grouped is based on the value of the identifier column (@Id), therefore, we strongly recommend using the @GeneratedValue annotation for the identifier, which ensures that the order of the increment identifier.

Sliding windows

In this case, unlike the previous case, the output grouped record is generated for each newly inserted record, and the calculation of group function values is carried out for some group of records, the size of which is determined using the keyword WINDOW BY column INTERVAL = value. It should be noted that the syntax of this keyword differs from the generally accepted one - the only column with values and the interval by which the window size is determined are set directly in WINDOW BY:

```
String sql = "select stream count(e.eventId), sum(e.eventValue) from
su.interference.test.entity.Event e window by e.eventId interval =
100";

StreamQueue q = (StreamQueue) s.execute(sql);
while (true) {
    Object o = q.poll();
    ...
}
```

If the WINDOW BY keyword contains a column marked with @Id annotation, then the window size (in the rows) will be constant and equal to the value specified in INTERVAL.

## Additional CEP features

Below, we will list the features used in event processing for understanding how it works within a cluster:

- SELECT STREAM always returns DIRTY READS data, regardless of which node the request is executed and whether the commit was executed
- session.purge(object) operation deletes an given object regardless of transaction state and available to execute only on the same node where data is inserted!
- @NoDistribute annotation disables remote data synchronization for cases when the event stream is processed on a single node and there is no need to save this data on other nodes.
- @Threshold annotation limits the maximum amount of data in the table; upon reaching a given threshold, data will be deleted from the beginning automatically

## Distributed locks

When you try to change the object with a specific Id on any node at the cluster level, it is ensured that the object is locked on the owner node, i.e. on the node where this object was entered into the database. In this regard, it is recommended that mass changes be made on the data owner node, which will exclude additional remote locking requests. By default, the ability to change data from nodes that are not owners thru persist() operations is disabled. You can enable it by setting the distribute.lock = true configuration parameter (Not available in version 2021.1) or use PROCESS statement with execute() operation. The object lock in the cluster is released by commit or rollback.

## Fault Tolerance

In the normal cluster operation mode, each of the nodes regularly sends messages to the remaining nodes of the cluster.

If the connection between the current node and a remote node is interrupted, all identifiers of the changed frames that cannot be replicated to this node are saved until the functionality of this node is restored.

After restoration of the node's operability, the node checks the availability of all other nodes. The remaining nodes, in turn, roll forward to the offline node of the changes that occurred during its inoperability. After the roll of changes and subject to the availability of all other nodes, the node goes online.

## Performance

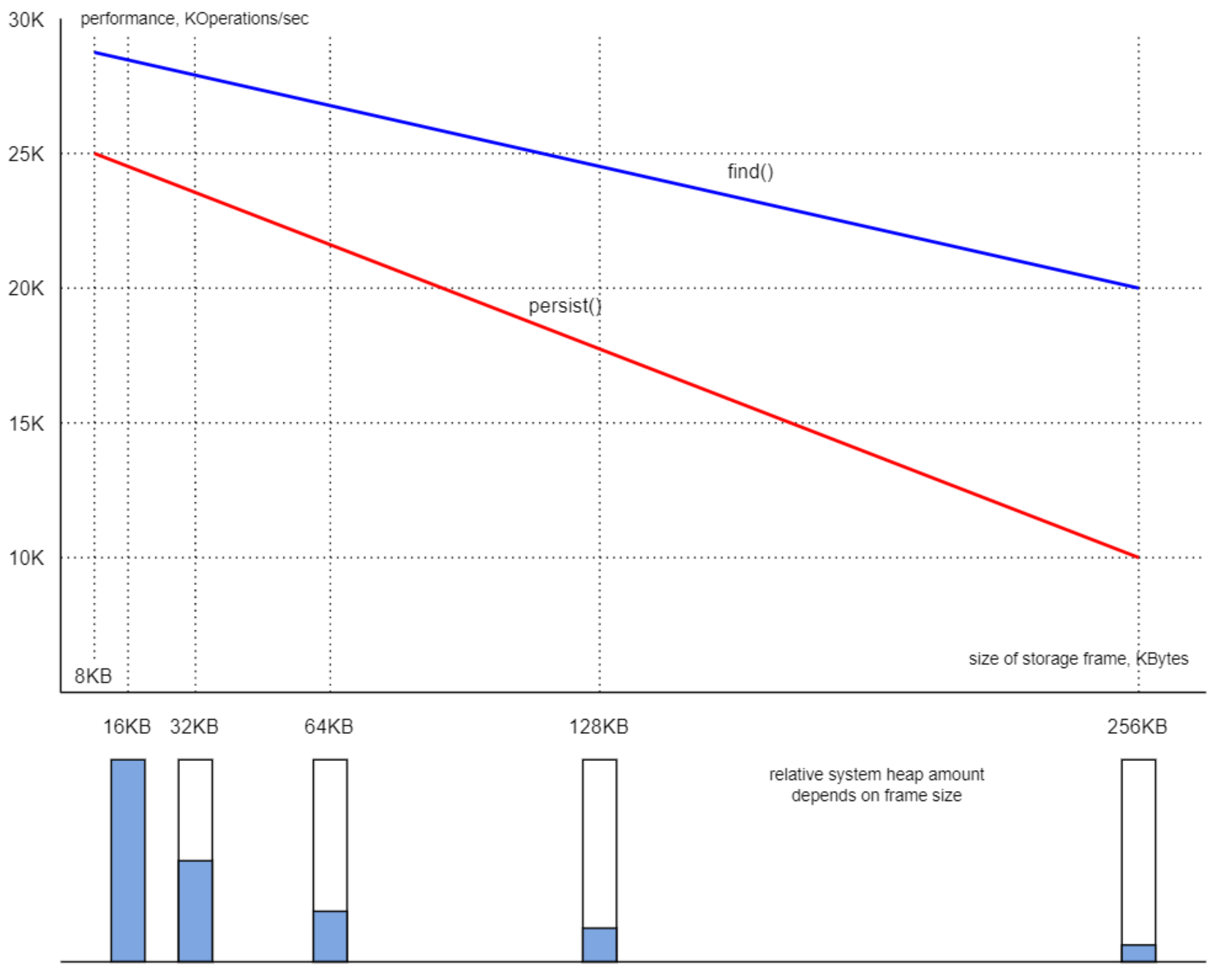
Usually, on modern hardware, the performance of inserting a separate java objects (events) into the persistent storage on a specific node can reach up to 100000 objects per second. In reality, performance is affected by many factors that somehow reduce this value:

- as a rule, the table contains at least one index field (@Id), often more than one, but inserting a record into the index is a separate insert action, therefore, inserting one object into some table with an index field will be slower than into simple table
- the increasing size of the frame will slightly reduce inserting performance, ususally, the overall performance of the inserts decreases linearly with the increase in the size of the frame (see picture below)
- the table size for tables with index fields, with an increase in the size of indexes, the performance of inserts and reads slightly reduce
- the size of the allocated heap memory, an larger heap size provides greater performance with large storage sizes, since reduces the frequency of clearing frames in memory
- inserts with ordered incremental values of the index field will be faster than with unordered (random) values

There is a definite relationship between frame size, heap size, and the overall performance of inserts and reads. As a rule, the best performance of data inserts is observed for small frame sizes, and we could stop there, but in practice, the internal mechanisms of the cluster use the definitions for frames constantly loaded into memory, as well as auxiliary data for transactions and transport communications. For smaller frame sizes, the more objects will be on the heap for a given storage size and the more memory we need. It is also critical for good insert performance to have the largest possible number of index frames directly loaded into memory. If the heap is small relative to the storage size, the index frames will be flushed from the heap more frequently and re-read from disk as needed, which degrades performance. Here we will need to determine some trade-off between the speed of inserts and the amount of RAM that we can allocate for the heap. A relatively detailed picture of these dependencies is shown in the graph, roughly speaking, if our heap size is not limited, then we can use the minimum values for the frame size, but if there are limitations, then they must be taken into account when planning the storage size and the planned number of new records in some unit of time.

The performance values of inserts (persist) and reads (find by id) in the graph were obtained on a cheap Mac Mini Late 2012 i7 2.3 16G RAM 1TB SSD on a small storage size (near to 100GB) by measuring 5000000 inserts and reads into/from a table with a record size of 3 fields and one index field (@Id) with an incremental filling of the identifier and using the @NoCheck annotation. Heap cleanup is enabled. Xmx8G.

As you can see from the graph, the size of frames has more little effect on the performance of readings.



## Configuration parameters

The current configuration is contained in the config/interference.properties file

The following describes the values of the configuration parameters and provides the optimal values for most applications of the parameters:

**local.node.id** - node identifier in the cluster,
Integer value from 1 to 64.
All nodes in the cluster must have unique identifiers.
The parameter must be specified when creating the instance and cannot be changed further.

**files.amount** - the number of threads that have the ability to simultaneously execute changes to the repository. Each thread operates with own, unique selected file. It is recommended to set a value equal to the number of processor cores.
The default value is 4.
Values from 1 to 64 can be used.
The parameter must be specified when creating the instance and cannot be changed further.

**frame.size** - size of the physical storage frame in bytes.
The default value is 16384.
Values 4096, 16384, 32768, 65536, 131072, 262144, 524288 can be used only. Higher values reduce the size of the internal system data in heap but may decrease performance, see Performance section.
The parameter must be specified when creating the instance and cannot be changed further.

**frame.size.ix** - size of the physical frame for storing indexes (in bytes).
The default value is 16384.
Values 4096, 16384, 32768, 65536, 131072, 262144, 524288 can be used only. Higher values reduce the size of the internal system data in heap but may decrease performance, see Performance section.
The parameter must be specified when creating the instance and cannot be changed further.

**codepage** - The codepage used to serialize string objects (String).
The default value is UTF-8.
The parameter must be specified when creating the instance and cannot be changed further.

**dateformat** - Date format used in SQL queries for string constants which used in WHERE clause condition, and, optionally in the management console (not avallable in 2021.1)

**db.path** - path to store data files

**journal.path** - path to store the journal file (not use in 2021.1)

**rmport** – initial numeric value which defining first server port for cluster transport interactions (see cluster configuration rules below).

**mmport** – http port for access to the control console via http protocol (not available in 2021.1). The parameter must be specified when creating the instance and cannot be changed further.

**diskio.mode** - write mode to disk.
2 values are used:
- **sync** (write through mode)
- **async** (write back mode).
By default, sync is used and it is not recommended to change it.

**sync.period** - time between writes of changed frames from the queue to disk in milliseconds.
The default value is 2000.
For OLTP systems, it is recommended to set it to 100-1000, for storages with rare changes - at 5000-10000.
Min value = 10, max value = 60000.

**sync.lock.enable** - lock data changes for the duration of a scheduled sync of frames to disk.
May be **true** or **false**.
By default, set to true.

**cluster.nodes** - list of nodeIds, hosts and ports of cluster nodes, separated by commas. The list must contains string of the following format:

nodeId:host:port,nodeId:host:port, ... etc.

If the value is not set, the node will function in single mode (as local database)

**retrieve.threads.amount** - the number of threads for parallel processing of the SQL query.

**retrieve.queue.size** – size (amount of elements) of blocking queue, which use in SQL retrieve mechanism for prevent of heap overload. For best performance, use max possibly value depends of your heap size and amount of SQL queries running simultaneously.
Usually, 10000-100000 is enough or optimal performance.

**auto.class.register** - a list of fully qualified names of entity classes, separated by commas, for which when the service starts, verification will be performed and, if necessary, automatic registration (both for services operating in standalone mode and at the application level)

**hbeat.period** - heartbeat period in milliseconds (not use in 2021.1).

**transport.sync.timeout** – internal transport inter-node synchronization timeout (in milliseconds)

**transport.read.buffer** – amount (in bytes) of internal transport read buffer

**transport.write.buffer** – amount (in bytes) of internal transport write buffer

**cleanup.enable** - enable heap management/cleanup
may be **true** or **false**, **true** by default. If **false** is described, then all frames loaded into heap memory will remain there until the instance terminates.

**cleanup.tx.timeout** – timeout in milleseconds for asynchronous cleanup internal transaction data for closed transactions

**cleanup.frames.timeout** – timeout for cleanup of unused frames in heap

**cleanup.data.threshold** - maximum amount of data frames in table before which the table will be excluded from the cleanup process

**cleanup.ix.threshold** - maximum amount of index frames before which the index will be excluded from the cleanup process

**cleanup.heap.data.threshold** - threshold in percent of heap usage upon reaching which all frames of data will be cleaned from heap memory (if they satisfy other conditions)

**cleanup.heap.ix.threshold** – same for indexes

**cleanup.heap.temp.threshold** – same for temp tables

**cleanup.heap.undo.threshold** – same for undo frames

## Cluster configuration rules

As we point above, interference open cluster use specific messages for provide interaction between cluster nodes. Each any two nodes in cluster uses in each of two directions statically configured transport channel (client -> server), which provide message delivery from one node to other. Each transport channel is one-directional, so, we need two configured transport channels for full interaction between two nodes. Therefore, several event servers should run on each node, the number of which is equal to the amount of cluster nodes – 1 (or total amount of parts of **cluster.nodes** parameter, which amount should be the same on each node). **Rmport** configuration parameter contains start value of server ports on current node, all additional server ports values calculated incrementally. In configuration this may described by next example (three nodes cluster with ip addresses = 192.168.100.1, 192.168.100.2, 192.168.100.3):

```
local.node.id=1
rmport=8050
cluster.nodes=2:192.168.100.2:8060,3:192.168.100.3:8070

local.node.id=2
rmport=8060
cluster.nodes=1:192.168.100.1:8050,3:192.168.100.3:8071

local.node.id=3
rmport=8070
cluster.nodes=1:192.168.100.1:8051,2:192.168.100.2:8061
```

As we see, each port may use only one time.

## Persistent data types

In the current version, interference open cluster supports the following persistent data types, which can be used as data types of fields of Entity classes (transient fields can be use any type):

```
int
long
float
double
java.lang.Integer
java.lang.Long
java.lang.Float
java.lang.Double
java.util.concurrent.AtomicInteger
java.util.concurrent.AtomicLong
java.lang.String
java.util.Date
java.util.ArrayList using the above types
java.util.HashMap using the above types
[] using the above types
```