JPA HIBERNATE PLAYGROUND PERFORMANCE

GERARD OLENSKI

1 ENTITY ID GENERATION STRATEGIES

2 PERSISTING IN BULKS

3 PERSISTING IN BATCHES

4 JPA DELETING METHODS

5 MANY-TO-ONE vs ONE-TO-MANY

ENTITY ID GENERATION STRATEGIES

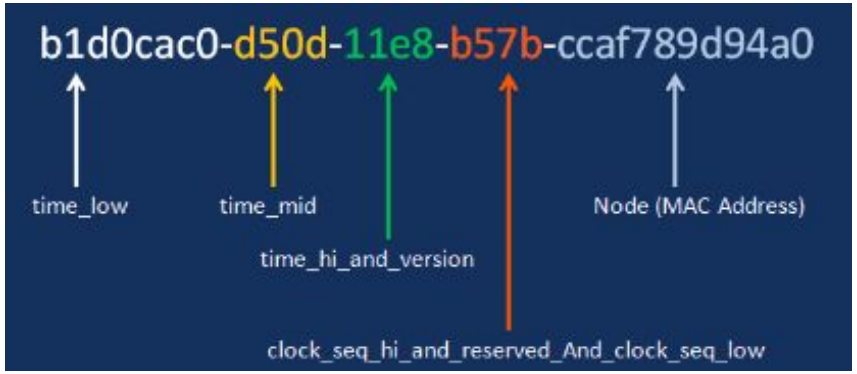
ID TYPES

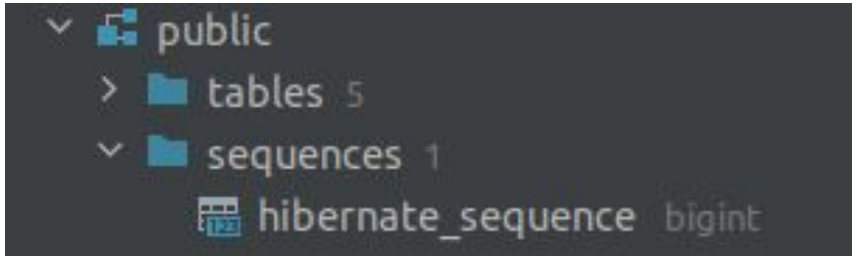
Two common ID types:

- UUID – RFC 4122 128 bit sequence
  - Postgres – UUID type
  - Oracle RAW(16)
- Numeric (e.g. long 64 bit two's complement integer)

Generation strategies:

- UUID – can be generated by application itself:
  - By application logic
  - Autogenerated By Hibernate
- Numeric – must be generated by DB
  - e.g. by calling sequence





UUID GENERATED BY APP LOGIC

```
class PostEntity {

    @Id
    @Column(name = "id")
    private UUID id;
```

```
PostEntity(@NonNull String author, @NonNull String category, @NonNull Str
    var now :LocalDateTime = LocalDateTime.now();
    this.id = randomUUID();
    this.creationDate = now.toLocalDate();
```

```
@Transactional
public void generate(int amount, Consumer<DbTime> dbProcessingTimeCallback) {
    log.info("Generating posts: amount={}", amount);
    var posts :Set<PostEntity> = rangeClosed(1, amount) IntStream
            .mapToObj(this::generatePost) Stream<PostEntity>
            .collect(toSet());

    log.info("Persisting posts");
    var watch :StopWatch = StopWatch.createStarted();
    postRepository.saveAll(posts);
```

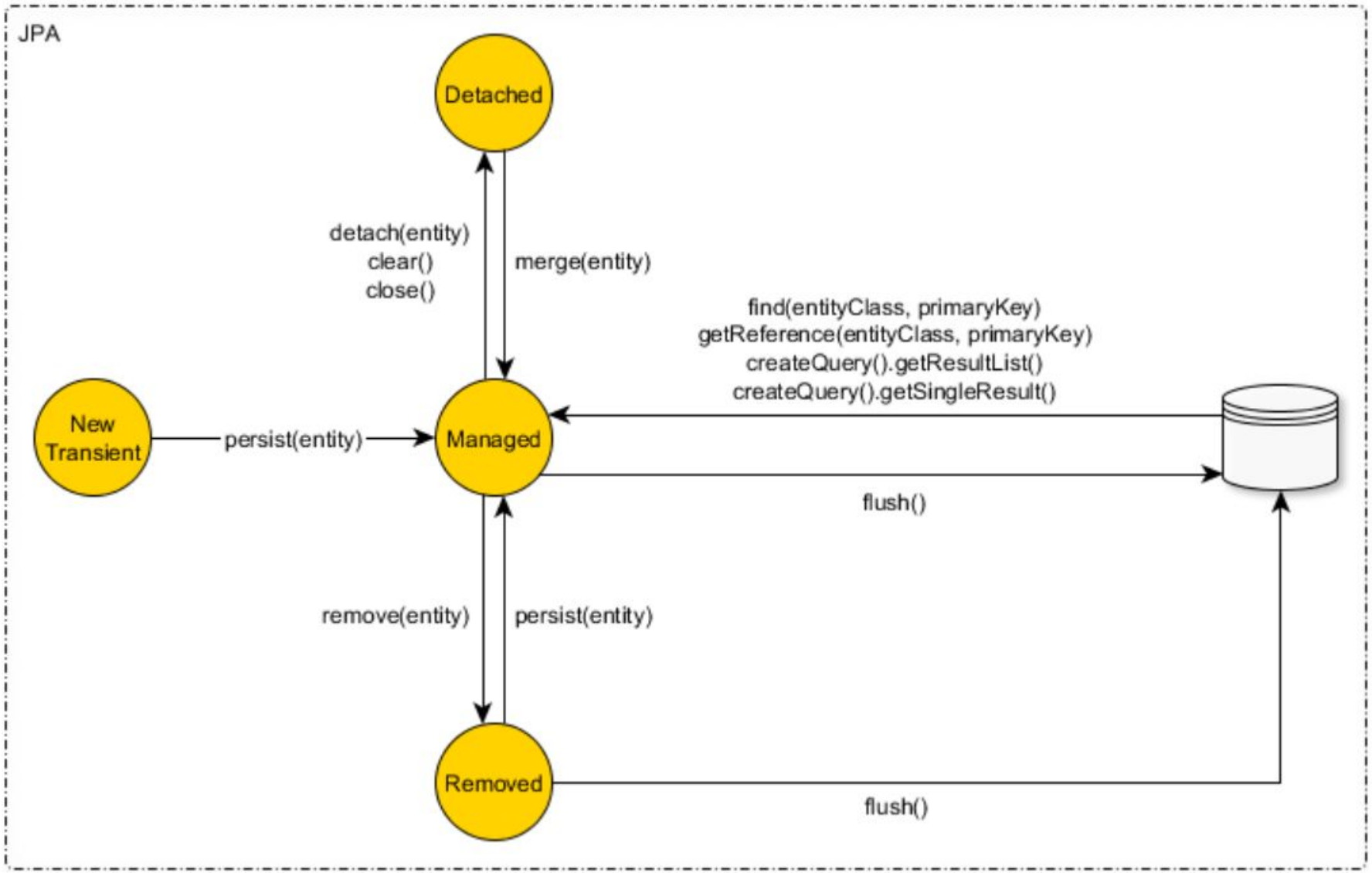
UUID GENERATED BY APP LOGIC – IS STH STRANGE?

```
2022-11-07 14:06:14.416  INFO 162678 --- [           main] o.g.j.infrastructure.post.PostGenerator  : Generating posts: amount=5
2022-11-07 14:06:14.422  INFO 162678 --- [           main] o.g.j.infrastructure.post.PostGenerator  : Persisting posts
Hibernate: select postentity0_.id as id1_0_0_, postentity0_.author as author2_0_0_, postentity0_.category as category3_0_0_, postenti
Hibernate: select postentity0_.id as id1_0_0_, postentity0_.author as author2_0_0_, postentity0_.category as category3_0_0_, postenti
Hibernate: select postentity0_.id as id1_0_0_, postentity0_.author as author2_0_0_, postentity0_.category as category3_0_0_, postenti
Hibernate: select postentity0_.id as id1_0_0_, postentity0_.author as author2_0_0_, postentity0_.category as category3_0_0_, postenti
Hibernate: select postentity0_.id as id1_0_0_, postentity0_.author as author2_0_0_, postentity0_.category as category3_0_0_, postenti
Hibernate: insert into post (author, category, content, creation_date, creation_time, topic, id) values (?, ?, ?, ?, ?, ?, ?)
Hibernate: insert into post (author, category, content, creation_date, creation_time, topic, id) values (?, ?, ?, ?, ?, ?, ?)
Hibernate: insert into post (author, category, content, creation_date, creation_time, topic, id) values (?, ?, ?, ?, ?, ?, ?)
Hibernate: insert into post (author, category, content, creation_date, creation_time, topic, id) values (?, ?, ?, ?, ?, ?, ?)
Hibernate: insert into post (author, category, content, creation_date, creation_time, topic, id) values (?, ?, ?, ?, ?, ?, ?)
2022-11-07 14:06:14.493  INFO 162678 --- [           main] i.StatisticalLoggingSessionEventListener : Session Metrics {
    537314 nanoseconds spent acquiring 1 JDBC connections;
    0 nanoseconds spent releasing 0 JDBC connections;
    619821 nanoseconds spent preparing 10 JDBC statements;
    8760747 nanoseconds spent executing 10 JDBC statements;
```

UUID GENERATED BY APP LOGIC – IS STH STRANGE?

```
2022-11-07 14:06:14.416  INFO 162678 --- [           main] o.g.j.infrastructure.post.PostGenerator  : Generating posts: amount=5
2022-11-07 14:06:14.422  INFO 162678 --- [           main] o.g.j.infrastructure.post.PostGenerator  : Persisting posts
Hibernate: select postentity0_.id as id1_0_0_, postentity0_.author as author2_0_0_, postentity0_.category as category3_0_0_, postenti
Hibernate: select postentity0_.id as id1_0_0_, postentity0_.author as author2_0_0_, postentity0_.category as category3_0_0_, postenti
Hibernate: select postentity0_.id as id1_0_0_, postentity0_.author as author2_0_0_, postentity0_.category as category3_0_0_, postenti
Hibernate: select postentity0_.id as id1_0_0_, postentity0_.author as author2_0_0_, postentity0_.category as category3_0_0_, postenti
Hibernate: select postentity0_.id as id1_0_0_, postentity0_.author as author2_0_0_, postentity0_.category as category3_0_0_, postenti
Hibernate: insert into post (author, category, content, creation_date, creation_time, topic, id) values (?, ?, ?, ?, ?, ?, ?)
Hibernate: insert into post (author, category, content, creation_date, creation_time, topic, id) values (?, ?, ?, ?, ?, ?, ?)
Hibernate: insert into post (author, category, content, creation_date, creation_time, topic, id) values (?, ?, ?, ?, ?, ?, ?)
Hibernate: insert into post (author, category, content, creation_date, creation_time, topic, id) values (?, ?, ?, ?, ?, ?, ?)
Hibernate: insert into post (author, category, content, creation_date, creation_time, topic, id) values (?, ?, ?, ?, ?, ?, ?)
2022-11-07 14:06:14.493  INFO 162678 --- [           main] i.StatisticalLoggingSessionEventListener : Session Metrics {
    537314 nanoseconds spent acquiring 1 JDBC connections;
    0 nanoseconds spent releasing 0 JDBC connections;
    619821 nanoseconds spent preparing 10 JDBC statements;
    8760747 nanoseconds spent executing 10 JDBC statements;
```

WHY IT HAPPENS?



HOW TO SOLVE IT?

- Add @Version property
- Implement Persistable interface
- Change to @GeneratedValue

```java
/**
 * Versioning supports optimistic locks.
 */
@Version
@Column(name = "version")
private Integer version;
```

```java
public abstract class BasePersistableEntity implements Persistable<UUID> {

    @Getter
    @Transient
    private boolean isNew = true;

    Gerard Olenski
    @PrePersist
    @PostLoad
    void markNotNew() { this.isNew = false; }
}
```

```java
@Id
@GeneratedValue
@Column(name = "id")
private UUID id;
```

UUID GENERATED BY APP LOGIC – FIXED

```
2022-11-07 15:29:32.599  INFO 168916 --- [           main] o.g.j.i.post.PersistablePostGenerator    : Generating posts: amount=5
2022-11-07 15:29:32.606  INFO 168916 --- [           main] o.g.j.i.post.PersistablePostGenerator    : Persisting posts
Hibernate: insert into post_persist (author, category, content, creation_date, creation_time, topic, id) values (?, ?, ?, ?, ?, ?, ?)
Hibernate: insert into post_persist (author, category, content, creation_date, creation_time, topic, id) values (?, ?, ?, ?, ?, ?, ?)
Hibernate: insert into post_persist (author, category, content, creation_date, creation_time, topic, id) values (?, ?, ?, ?, ?, ?, ?)
Hibernate: insert into post_persist (author, category, content, creation_date, creation_time, topic, id) values (?, ?, ?, ?, ?, ?, ?)
Hibernate: insert into post_persist (author, category, content, creation_date, creation_time, topic, id) values (?, ?, ?, ?, ?, ?, ?)
2022-11-07 15:29:32.655  INFO 168916 --- [           main] i.StatisticalLoggingSessionEventListener : Session Metrics {
    608609 nanoseconds spent acquiring 1 JDBC connections;
    0 nanoseconds spent releasing 0 JDBC connections;
    255110 nanoseconds spent preparing 5 JDBC statements;
    13454583 nanoseconds spent executing 5 JDBC statements;
```

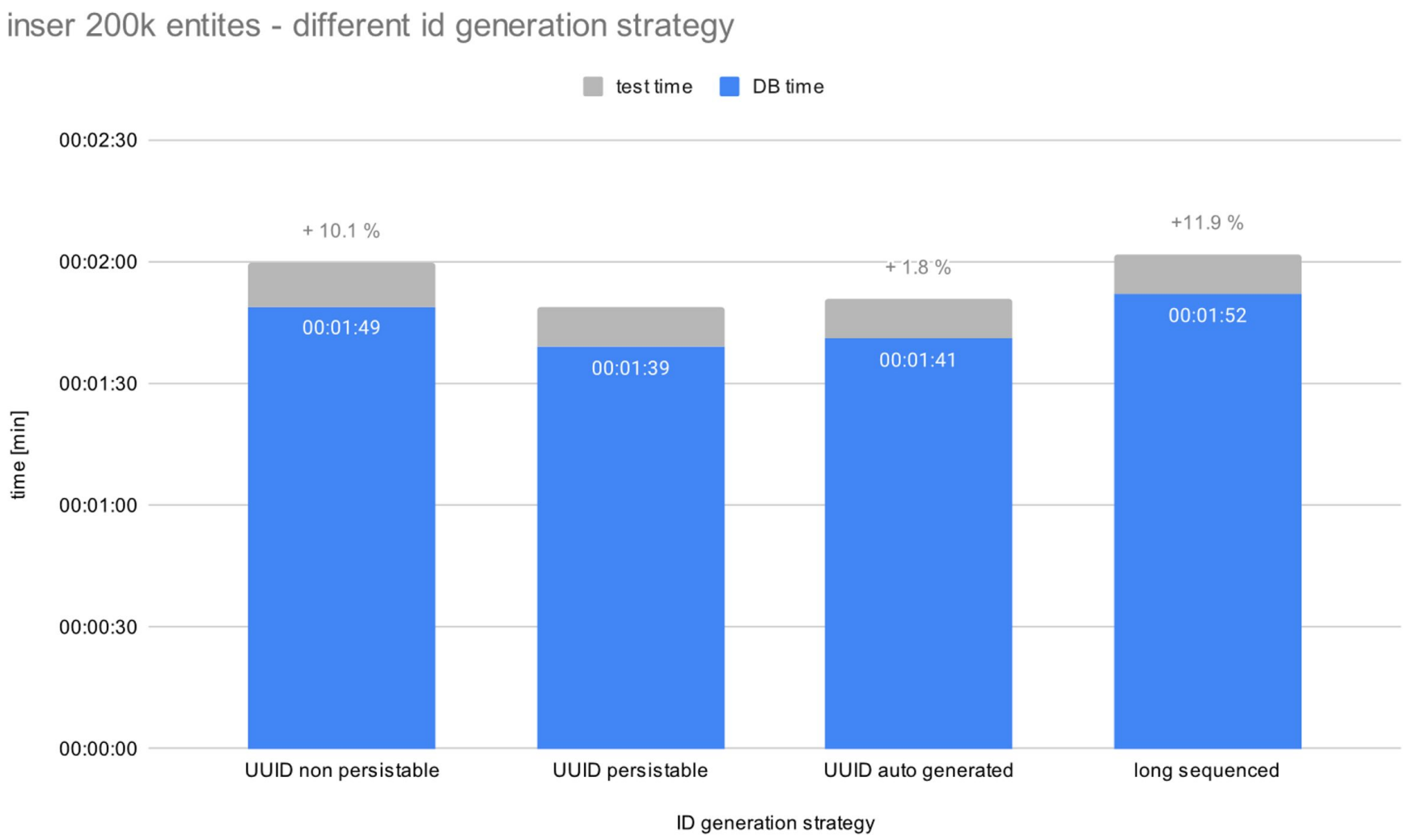
LONG AUTO GENERATED

```
public
  tables 5
  sequences 1
    hibernate_sequence bigint
```

```java
@Id
@GeneratedValue(strategy = SEQUENCE)
@Column(name = "id")
private long id;
```

```
2022-11-07 15:31:55.515  INFO 169207 --- [           main] o.g.j.i.post.SequencedIdPostGenerator    : Generating posts: amount=5
2022-11-07 15:31:55.522  INFO 169207 --- [           main] o.g.j.i.post.SequencedIdPostGenerator    : Persisting posts
Hibernate: select nextval ('hibernate_sequence')
Hibernate: select nextval ('hibernate_sequence')
Hibernate: select nextval ('hibernate_sequence')
Hibernate: select nextval ('hibernate_sequence')
Hibernate: select nextval ('hibernate_sequence')
Hibernate: insert into post_seq_id (author, category, content, creation_date, creation_time, topic, id) values (?, ?, ?, ?, ?, ?, ?)
Hibernate: insert into post_seq_id (author, category, content, creation_date, creation_time, topic, id) values (?, ?, ?, ?, ?, ?, ?)
Hibernate: insert into post_seq_id (author, category, content, creation_date, creation_time, topic, id) values (?, ?, ?, ?, ?, ?, ?)
Hibernate: insert into post_seq_id (author, category, content, creation_date, creation_time, topic, id) values (?, ?, ?, ?, ?, ?, ?)
Hibernate: insert into post_seq_id (author, category, content, creation_date, creation_time, topic, id) values (?, ?, ?, ?, ?, ?, ?)
2022-11-07 15:31:55.579  INFO 169207 --- [           main] i.StatisticalLoggingSessionEventListener : Session Metrics {
    493164 nanoseconds spent acquiring 1 JDBC connections;
    0 nanoseconds spent releasing 0 JDBC connections;
    517180 nanoseconds spent preparing 10 JDBC statements;
    15992501 nanoseconds spent executing 10 JDBC statements;
```

HOW IT AFFECT PERFORMANCE



PERSISTING IN BULKS

PROBLEM SCOPE: PERSISTING LARGE AMOUNT OF ENTITIES

How to do it?

Simply persist each entity in loop?

```
begin;
insert into post values (?, ?, ?);
commit;
```

But what with performance?

So maybe persist all entities in one transaction?

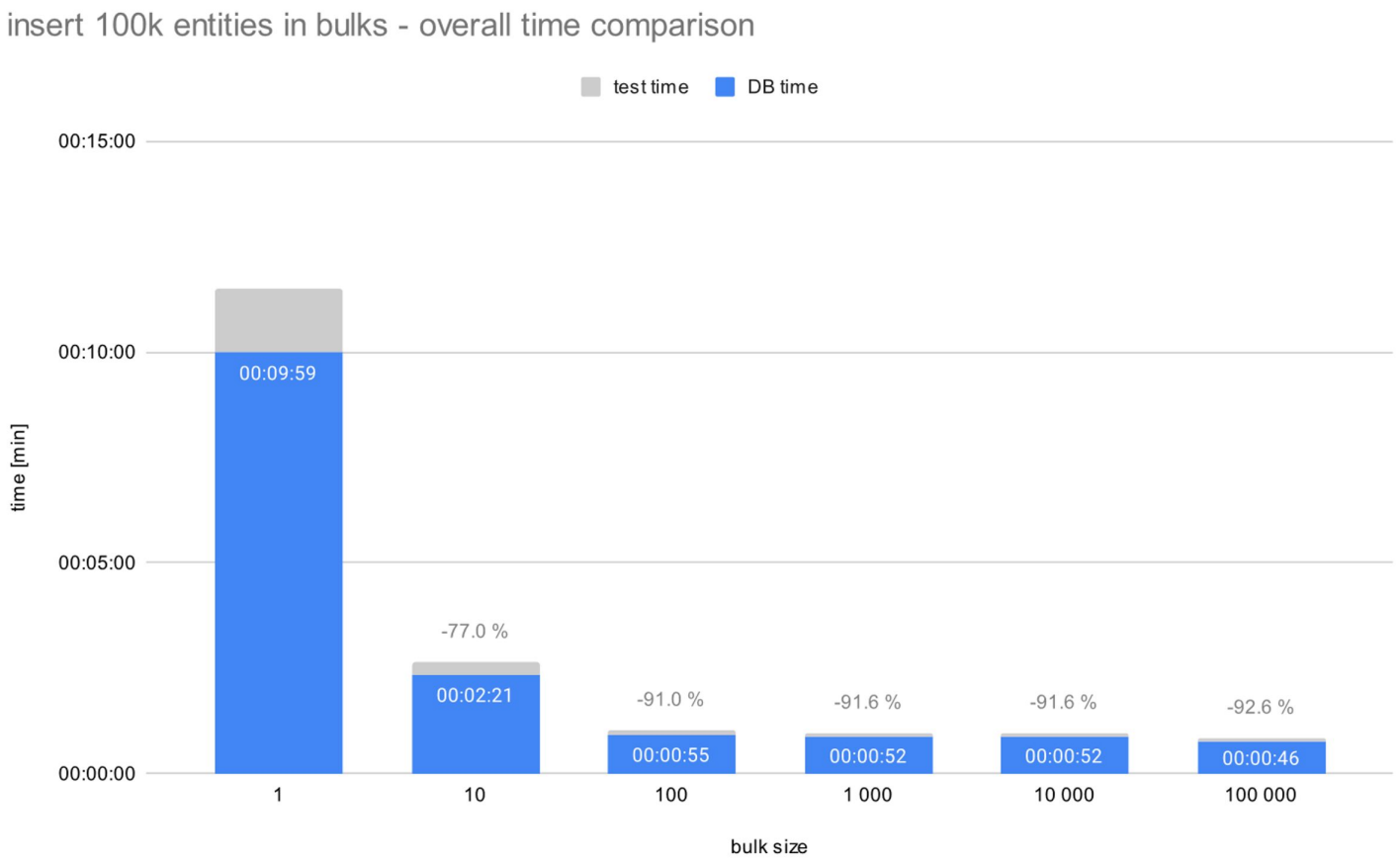
```
begin;
insert into post values (?, ?, ?);
insert into post values (?, ?, ?);
insert into post values (?, ?, ?);
commit;
```

But what with DB then: size of transaction log, possible rollbacks?

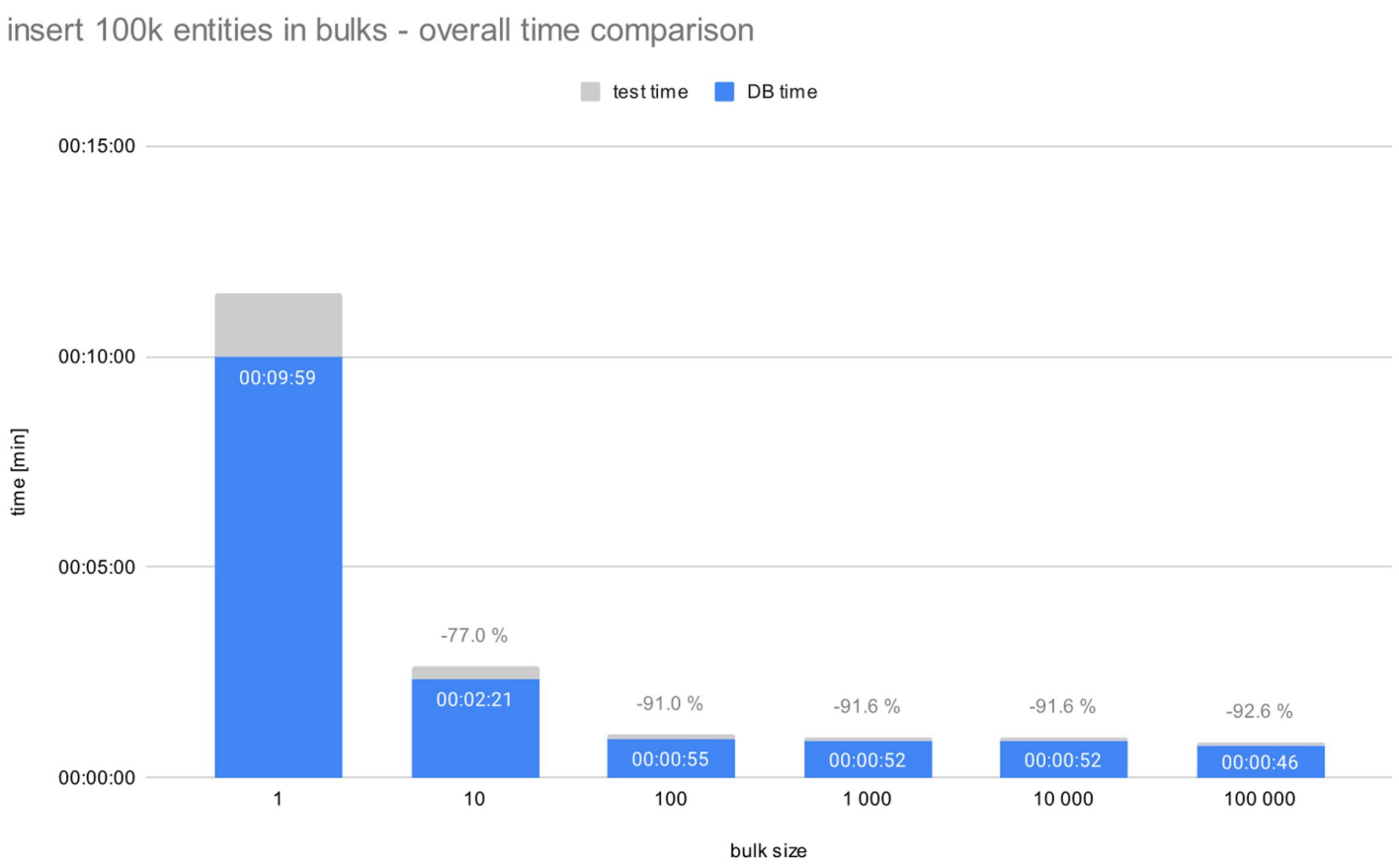
So maybe let's do it in bulks …

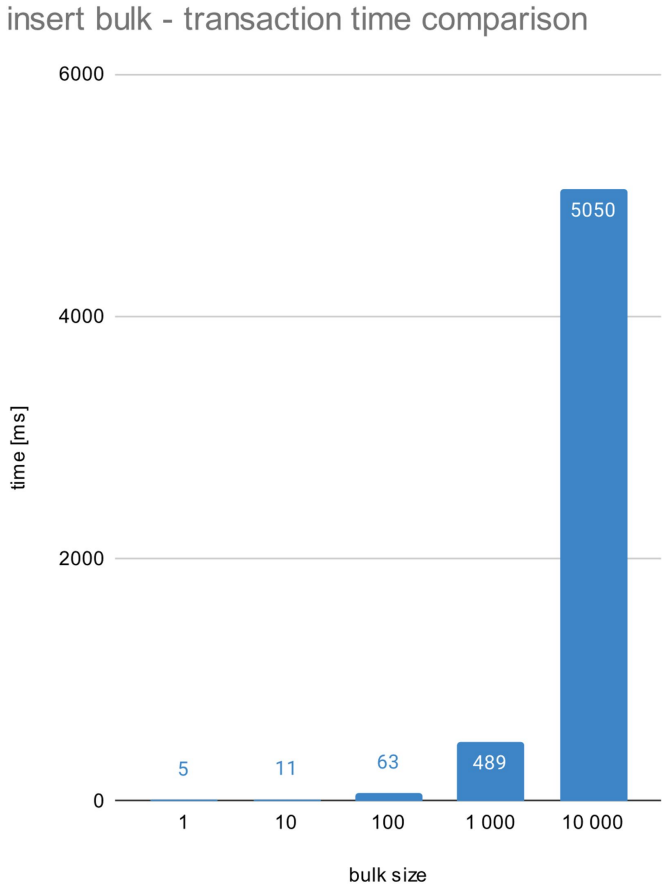
But how the bulk size affects performance?

HOW IT AFFECT PERFORMANCE



BUT CONSIDER ALSO TRANSACTION TIME





PERSISTING IN BATCHES

HOW TO IMPROVE BULK APPROACH

By default in hibernate, all entity insert statements inside one transaction go one by one. It can be seen easily just by turning on hibernate statistics logging: **`spring.jpa.properties.hibernate.generate_statistics=true`**

```
2022-11-08 17:16:52.982  INFO 302576 --- [           main] o.g.j.i.post.AutoIdPostGenerator         : Persisting posts
Hibernate: insert into post_auto_id (author, category, content, creation_date, creation_time, topic, id) values (?, ?, ?, ?, ?, ?, ?)
Hibernate: insert into post_auto_id (author, category, content, creation_date, creation_time, topic, id) values (?, ?, ?, ?, ?, ?, ?)
Hibernate: insert into post_auto_id (author, category, content, creation_date, creation_time, topic, id) values (?, ?, ?, ?, ?, ?, ?)
Hibernate: insert into post_auto_id (author, category, content, creation_date, creation_time, topic, id) values (?, ?, ?, ?, ?, ?, ?)
Hibernate: insert into post_auto_id (author, category, content, creation_date, creation_time, topic, id) values (?, ?, ?, ?, ?, ?, ?)
2022-11-08 17:16:53.033  INFO 302576 --- [           main] i.StatisticalLoggingSessionEventListener : Session Metrics {
    482574 nanoseconds spent acquiring 1 JDBC connections;
    0 nanoseconds spent releasing 0 JDBC connections;
    278541 nanoseconds spent preparing 5 JDBC statements;
    8778108 nanoseconds spent executing 5 JDBC statements;
    0 nanoseconds spent executing 0 JDBC batches;
```

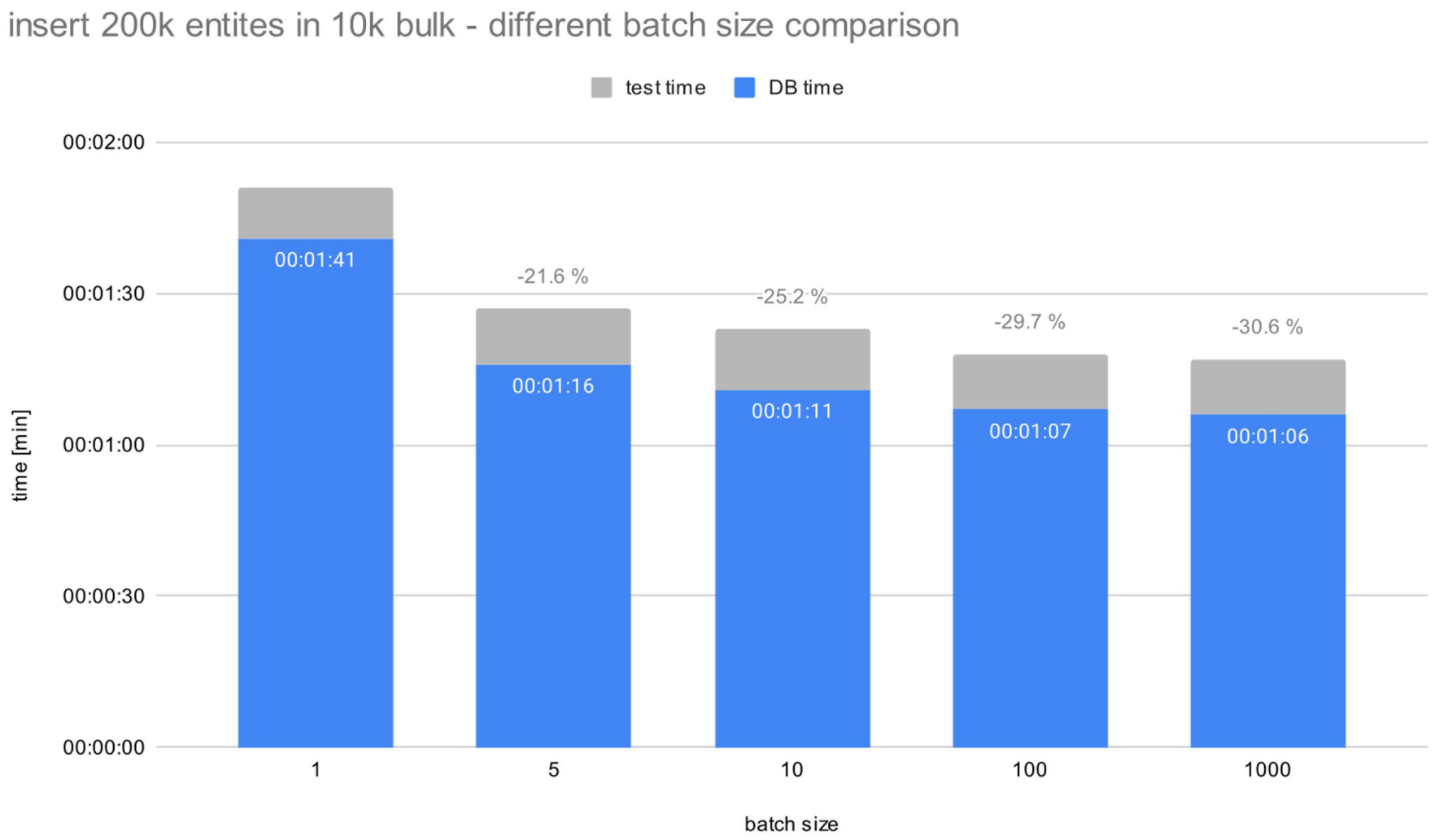
BUT SQL SUPPORTS BATCH INSERTS

```
begin;
insert into post values (?, ?, ?), (?, ?, ?), (?, ?, ?);
insert into post values (?, ?, ?), (?, ?, ?), (?, ?, ?);
insert into post values (?, ?, ?), (?, ?, ?), (?, ?, ?);
commit;
```

To enable batch inserts in hibernate, the `spring.jpa.properties.hibernate.jdbc.batch_size` property can be used.

```
2022-11-08 17:28:15.534  INFO 303631 --- [           main] o.g.j.i.post.AutoIdPostGenerator         : Persisting posts
Hibernate: insert into post_auto_id (author, category, content, creation_date, creation_time, topic, id) values (?, ?, ?, ?, ?, ?, ?)
Hibernate: insert into post_auto_id (author, category, content, creation_date, creation_time, topic, id) values (?, ?, ?, ?, ?, ?, ?)
Hibernate: insert into post_auto_id (author, category, content, creation_date, creation_time, topic, id) values (?, ?, ?, ?, ?, ?, ?)
Hibernate: insert into post_auto_id (author, category, content, creation_date, creation_time, topic, id) values (?, ?, ?, ?, ?, ?, ?)
Hibernate: insert into post_auto_id (author, category, content, creation_date, creation_time, topic, id) values (?, ?, ?, ?, ?, ?, ?)
Hibernate: insert into post_auto_id (author, category, content, creation_date, creation_time, topic, id) values (?, ?, ?, ?, ?, ?, ?)
Hibernate: insert into post_auto_id (author, category, content, creation_date, creation_time, topic, id) values (?, ?, ?, ?, ?, ?, ?)
Hibernate: insert into post_auto_id (author, category, content, creation_date, creation_time, topic, id) values (?, ?, ?, ?, ?, ?, ?)
Hibernate: insert into post_auto_id (author, category, content, creation_date, creation_time, topic, id) values (?, ?, ?, ?, ?, ?, ?)
Hibernate: insert into post_auto_id (author, category, content, creation_date, creation_time, topic, id) values (?, ?, ?, ?, ?, ?, ?)
2022-11-08 17:28:15.594  INFO 303631 --- [           main] i.StatisticalLoggingSessionEventListener : Session Metrics {
    446133 nanoseconds spent acquiring 1 JDBC connections;
    0 nanoseconds spent releasing 0 JDBC connections;
    142955 nanoseconds spent preparing 1 JDBC statements;
    0 nanoseconds spent executing 0 JDBC statements;
    18332012 nanoseconds spent executing 3 JDBC batches;
```

HOW IT AFFECT PERFORMANCE



JPA DELETING METHODS

DELETE ALL

`JpaRepository#deleteAll()`

DELETE ALL

`JpaRepository#deleteAll()`

- selects all entities
- deletes one by one

```
2022-11-18 19:16:05.658  INFO 10219 --- [           main] o.g.j.i.post.DeleteDataSqlLogTest        : Deleting entities
Hibernate: select autoidpost0_.id as id1_1_, autoidpost0_.author as author2_1_, autoidpost0_.category as category3_1_, autoidpost0
Hibernate: delete from post_auto_id where id=? and version=?
Hibernate: delete from post_auto_id where id=? and version=?
Hibernate: delete from post_auto_id where id=? and version=?
Hibernate: delete from post_auto_id where id=? and version=?
Hibernate: delete from post_auto_id where id=? and version=?
Hibernate: delete from post_auto_id where id=? and version=?
Hibernate: delete from post_auto_id where id=? and version=?
Hibernate: delete from post_auto_id where id=? and version=?
Hibernate: delete from post_auto_id where id=? and version=?
Hibernate: delete from post_auto_id where id=? and version=?
2022-11-18 19:16:05.695  INFO 10219 --- [           main] i.StatisticalLoggingSessionEventListener : Session Metrics {
    14038 nanoseconds spent acquiring 1 JDBC connections;
    0 nanoseconds spent releasing 0 JDBC connections;
    303047 nanoseconds spent preparing 11 JDBC statements;
    3714497 nanoseconds spent executing 11 JDBC statements;
    0 nanoseconds spent executing 0 JDBC batches;
```

DELETE ALL IN BATCH

`JpaRepository#deleteAllInBatch()`

DELETE ALL IN BATCH

`JpaRepository#deleteAllInBatch()`

- SQL: delete from entity

```
2022-11-18 19:26:19.068  INFO 15119 --- [           main] o.g.j.i.post.DeleteDataSqlLogTest        : Deleting entities
Hibernate: delete from post_auto_id
2022-11-18 19:26:19.078  INFO 15119 --- [           main] i.StatisticalLoggingSessionEventListener : Session Metrics {
    20302 nanoseconds spent acquiring 1 JDBC connections;
    0 nanoseconds spent releasing 0 JDBC connections;
    87932 nanoseconds spent preparing 1 JDBC statements;
    591203 nanoseconds spent executing 1 JDBC statements;
    0 nanoseconds spent executing 0 JDBC batches;
```

DELETE ALL BY ID

`JpaRepository#deleteAllById(Iterable<ID>)`

DELETE ALL BY ID

`JpaRepository#deleteAllById(Iterable<? Extends ID>)`

- first selects entire entity
- then deletes one by one

```
2022-11-18 19:34:41.257  INFO 18041 --- [           main] o.g.j.i.post.DeleteDataSqlLogTest        : Deleting entities
Hibernate: select autoidpost0_.id as id1_1_0_, autoidpost0_.author as author2_1_0_, autoidpost0_.category as category3_1_0_, autoidpo
Hibernate: select autoidpost0_.id as id1_1_0_, autoidpost0_.author as author2_1_0_, autoidpost0_.category as category3_1_0_, autoidpo
Hibernate: select autoidpost0_.id as id1_1_0_, autoidpost0_.author as author2_1_0_, autoidpost0_.category as category3_1_0_, autoidpo
Hibernate: select autoidpost0_.id as id1_1_0_, autoidpost0_.author as author2_1_0_, autoidpost0_.category as category3_1_0_, autoidpo
Hibernate: select autoidpost0_.id as id1_1_0_, autoidpost0_.author as author2_1_0_, autoidpost0_.category as category3_1_0_, autoidpo
Hibernate: delete from post_auto_id where id=? and version=?
Hibernate: delete from post_auto_id where id=? and version=?
Hibernate: delete from post_auto_id where id=? and version=?
Hibernate: delete from post_auto_id where id=? and version=?
Hibernate: delete from post_auto_id where id=? and version=?
2022-11-18 19:34:41.286  INFO 18041 --- [           main] i.StatisticalLoggingSessionEventListener : Session Metrics {
    20086 nanoseconds spent acquiring 1 JDBC connections;
    0 nanoseconds spent releasing 0 JDBC connections;
    511935 nanoseconds spent preparing 10 JDBC statements;
    4912805 nanoseconds spent executing 10 JDBC statements;
    0 nanoseconds spent executing 0 JDBC batches;
```

DELETE ALL BY ID IN BATCH

`JpaRepository#deleteAllByIdInBatch(Iterable<ID>)`

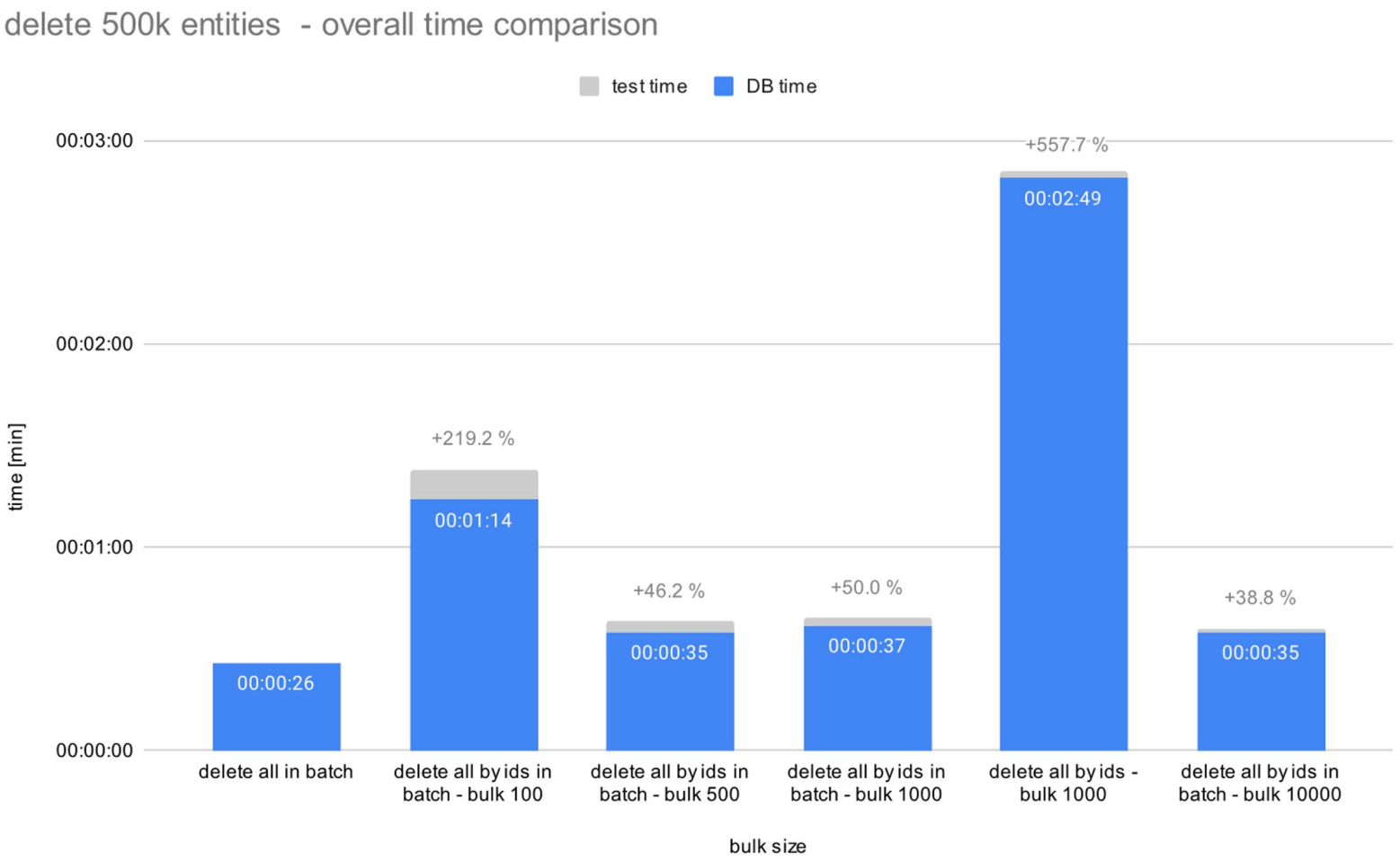
DELETE ALL BY ID IN BATCH

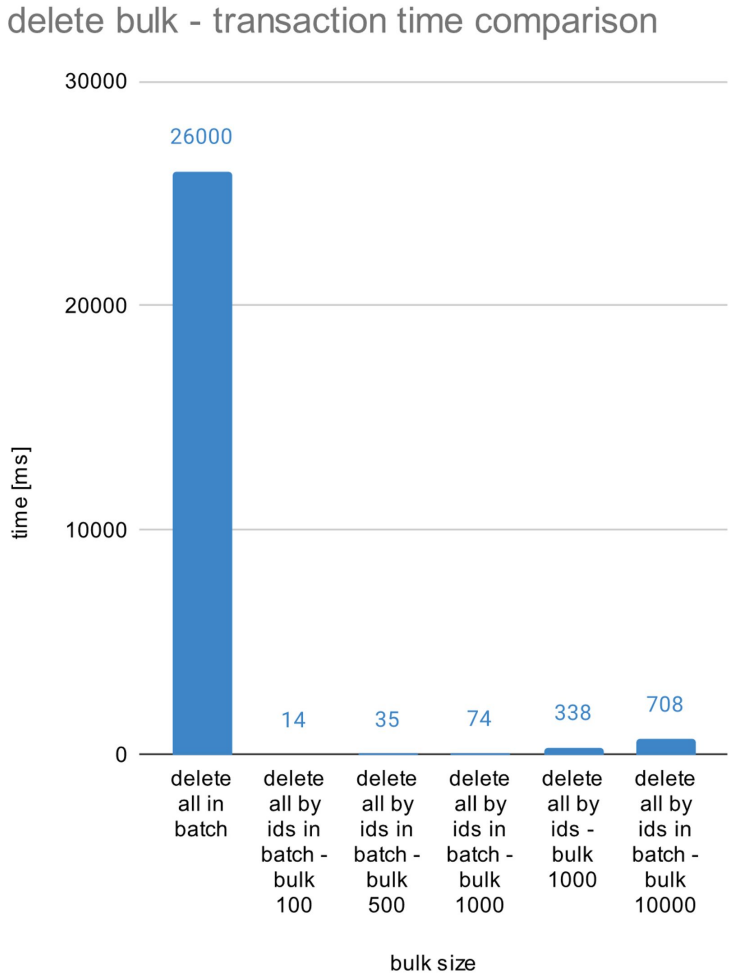
`JpaRepository#deleteAllByIdInBatch(Iterable<ID>)`

- deletes in one statement

```
2022-11-18 19:37:43.917  INFO 18398 --- [           main] o.g.j.i.post.DeleteDataSqlLogTest        : Deleting entities
Hibernate: delete from post_auto_id where id in (? , ? , ? , ? , ? , ? , ? , ? , ? , ?)
2022-11-18 19:37:43.934  INFO 18398 --- [           main] i.StatisticalLoggingSessionEventListener : Session Metrics {
    17736 nanoseconds spent acquiring 1 JDBC connections;
    0 nanoseconds spent releasing 0 JDBC connections;
    85318 nanoseconds spent preparing 1 JDBC statements;
    814667 nanoseconds spent executing 1 JDBC statements;
    0 nanoseconds spent executing 0 JDBC batches;
```

HOW IT AFFECT PERFORMANCE





DELETE ENTITIES IN BULK APPROACH

```
var bulk :Pageable = ofSize( pageSize: 1000);
Stream.generate(() -> queryAdapter.findIds(bulk))
        .takeWhile(Slice::hasContent)
        .forEach(ids -> cleanupAdapter.deleteAllByIdInBatch(ids));
```

Also consider foreign key with cascade delete.

```
@ManyToOne
@OnDelete(action = CASCADE)
@JoinColumn(name = "post_id", foreignKey = @ForeignKey(name = "pc_postId_fk"))
private PostMTOEntity post;
```

```
Hibernate: create table mto_post (id uuid not null, author varchar(30), category varchar(50), comment_count int8, content varchar(10000), creation_d
Hibernate: create table mto_post_comment (id uuid not null, author varchar(30), content varchar(10000), creation_date date, creation_time timestamp,
Hibernate: alter table if exists mto_post_comment add constraint pc_postId_fk foreign key (post_id) references mto_post on delete cascade
```

MANY-TO-ONE VS ONE-TO-MANY

FEW SUMMARIES (FROM HIGH-PERFORMANCE JAVA PERSISTENCE)

The foreign key is, therefore, the most important construct in building a table relationship, and, in a relation database, the child-side controls a table relationship.

Although `@OneToMany`, `@ManyToMany` or `@ElementCollection` are convenient from a data access perspective (entity state transitions can be cascaded from parent entities to children), they are definitely not free of cost. The price for reducing data access operations is paid in terms of result set fetching flexibility and performance. A JPA collection, either of entities or value types (basic or embeddables), binds a parent entity to a query that usually fetches all the associated child records. Because of this, the entity mapping becomes sensitive to the number of child entries.

FEW SUMMARIES (FROM HIGH-PERFORMANCE JAVA PERSISTENCE)

When handling large data sets, it is good practice to limit the result set size, both for UI (to increase responsiveness) or batch processing tasks (to avoid long running transactions). Just because JPA offers supports collection mapping, it does not mean they are mandatory for every domain model mapping. Until there is a clear understanding of the number of child records (or if there is even need to fetch child entities entirely), it is better to postpone the collection mapping decision. For high-performance systems, a data access query is often a much more flexible alternative.

Because the `@ManyToOne` association controls the foreign key directly, the automatically generated DML statements are very efficient.

Actually, the best-performing JPA associations always rely on the child-side to translate the JPA state to the foreign key column value.

This is one of the most important rules in JPA relationship mapping, and it will be further emphasized for `@OneToMany`, `@OneToOne` and even `@ManyToMany` associations.

FEW SUMMARIES (FROM HIGH-PERFORMANCE JAVA PERSISTENCE)

The bidirectional `@OneToMany` association generates efficient DML statements because the `@ManyToOne` mapping is in charge of the table relationship. Because it simplifies data access operations as well, the bidirectional `@OneToMany` association is worth considering when the size of the child records is relatively low.

Therefore, in reality, `@OneToMany` is practical only when many means few. Maybe `@OneToFew` would have been a more suggestive name for this annotation.

TESTED ENTITIES

```java
@OneToMany(mappedBy = "post", cascade = ALL, orphanRemoval = true)
private Set<PostCommentOTMEntity> comments = new HashSet<>();
```

```java
@ManyToOne
@JoinColumn(name = "post_id", foreignKey = @ForeignKey(name = "pc_postId_fk"))
private PostOTMEntity post;
```

ONE-TO-MANY INSERT

```
SELECT * FROM post WHERE id=?
SELECT * FROM post_comment WHERE post_id=?
INSERT INTO post_comment VALUES (?, ? ...)
UPDATE post WHERE id=? AND version=?
```

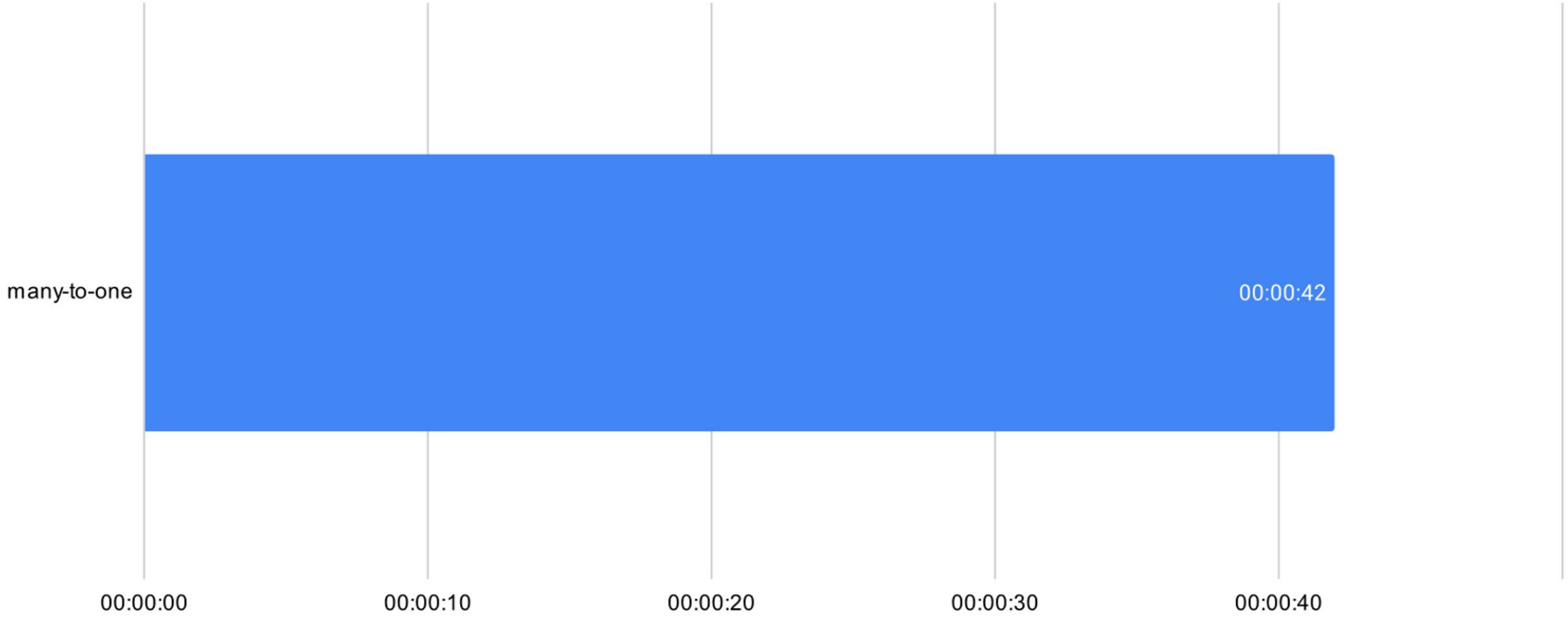
```
Hibernate: select postotment0_.id as id1_2_0_, postotment0_.author as author2_2_0_, postotment0_.category as category3_2_0_, postotment0_
  .comment_count as comment_4_2_0_, postotment0_.content as content5_2_0_, postotment0_.creation_date as creation6_2_0_, postotment0_.creation_time
   as creation7_2_0_, postotment0_.topic as topic8_2_0_, postotment0_.version as version9_2_0_ from otm_post postotment0_ where postotment0_.id=?
Hibernate: select comments0_.post_id as post_id7_3_0_, comments0_.id as id1_3_0_, comments0_.id as id1_3_1_, comments0_.author as author2_3_1_,
  comments0_.content as content3_3_1_, comments0_.creation_date as creation4_3_1_, comments0_.creation_time as creation5_3_1_, comments0_.post_id
  as post_id7_3_1_, comments0_.version as version6_3_1_ from otm_post_comment comments0_ where comments0_.post_id=?
Hibernate: insert into otm_post_comment (author, content, creation_date, creation_time, post_id, version, id) values (?, ?, ?, ?, ?, ?, ?)
Hibernate: update otm_post set author=?, category=?, comment_count=?, content=?, creation_date=?, creation_time=?, topic=?, version=? where id=?
  and version=?
```

MANY-TO-ONE INSERT

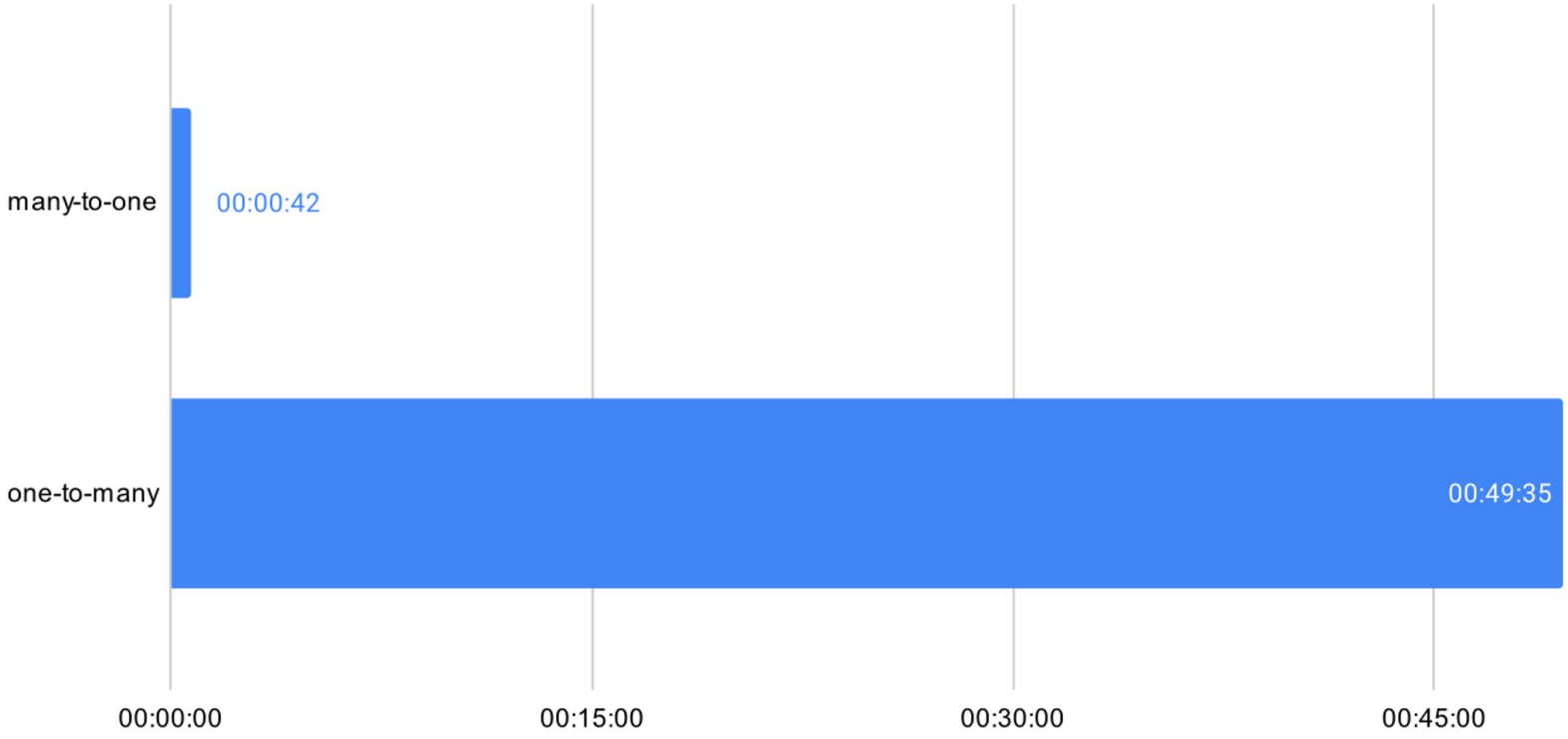
```
SELECT * FROM post WHERE id=?
INSERT INTO post_comment VALUES (?, ? ...)
UPDATE post WHERE id=? AND version=?
```

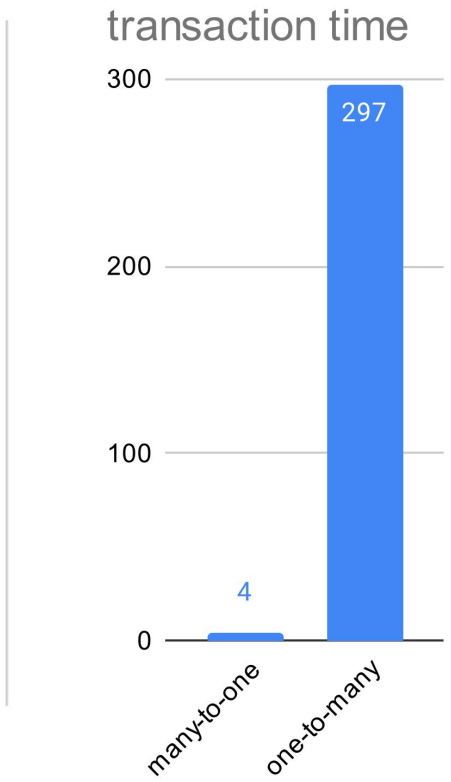
```
Hibernate: select postmtoent0_.id as id1_0_0_, postmtoent0_.author as author2_0_0_, postmtoent0_.category as category3_0_0_, postmtoent0_
 .comment_count as comment_4_0_0_, postmtoent0_.content as content5_0_0_, postmtoent0_.creation_date as creation6_0_0_, postmtoent0_.creation_time
  as creation7_0_0_, postmtoent0_.topic as topic8_0_0_, postmtoent0_.version as version9_0_0_ from mto_post postmtoent0_ where postmtoent0_.id=?
Hibernate: insert into mto_post_comment (author, content, creation_date, creation_time, post_id, version, id) values (?, ?, ?, ?, ?, ?, ?)
Hibernate: update mto_post set author=?, category=?, comment_count=?, content=?, creation_date=?, creation_time=?, topic=?, version=? where id=?
 and version=?
```

INSERT 10K COMMENTS

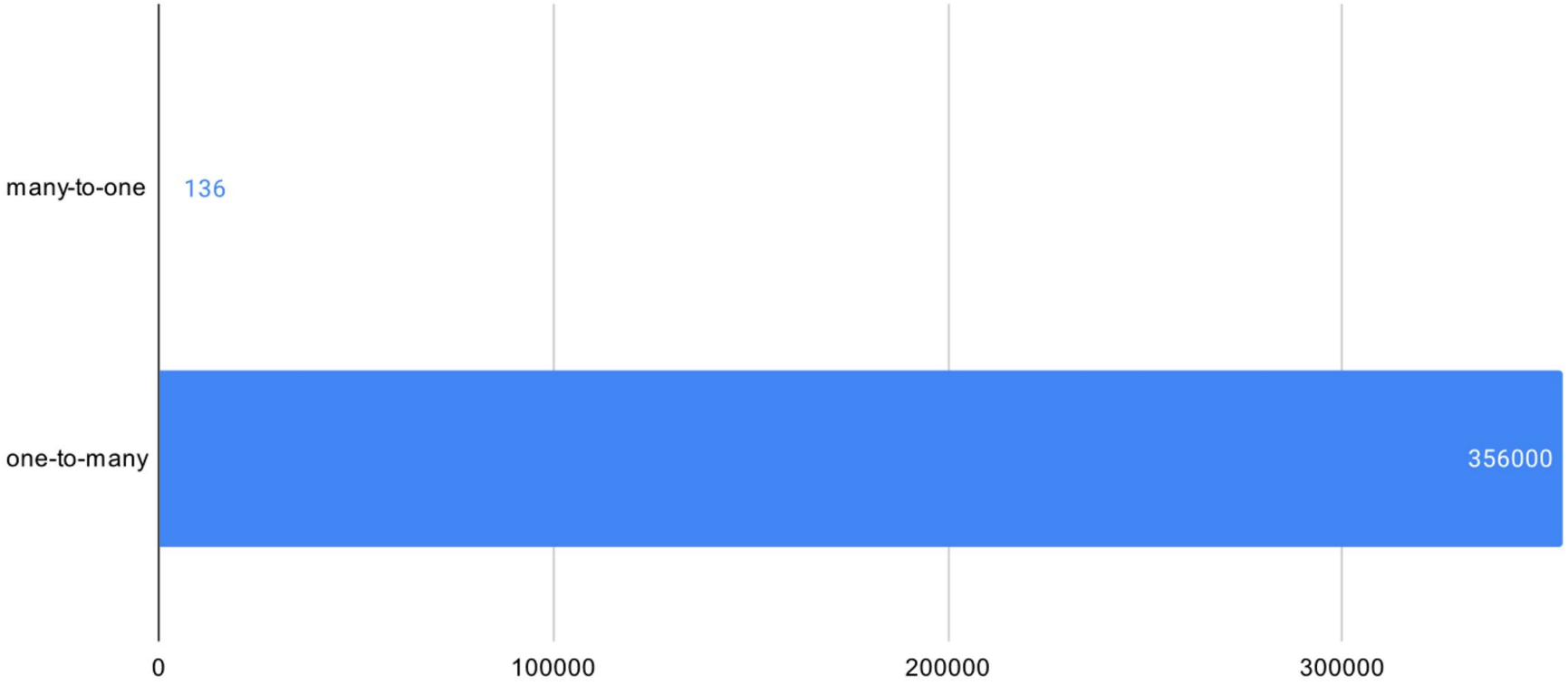


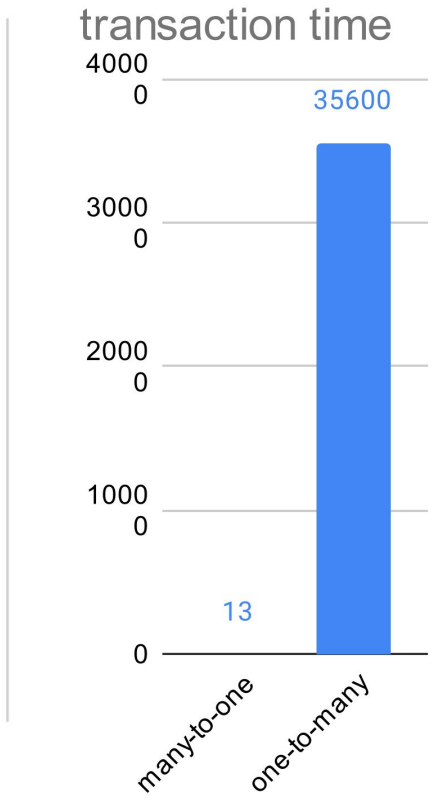
INSERT 10K COMMENTS





INSERT 10 NEW COMMENTS TO EXISTING 500K





BIBLIOGRAPHY

- *"High-Performance Java Persistence" Vlad Mihalcea*
- *https://vladmihalcea.com/tutorials/hibernate/*
- *https://docs.spring.io/spring-data/jpa/docs/current/reference/html/*
- *https://www.baeldung.com/category/persistence*
- *jpa-samples project: https://bitbucket.harman.com/users/golenski/repos/jpa-samples/browse*