



JOOQ  
Joy of SQL

# Kevin DAVIN

@davinkevin



Google Developer Expert  
on **Google Cloud & Kotlin**



Gitlab Heroes



Open Source Contributor



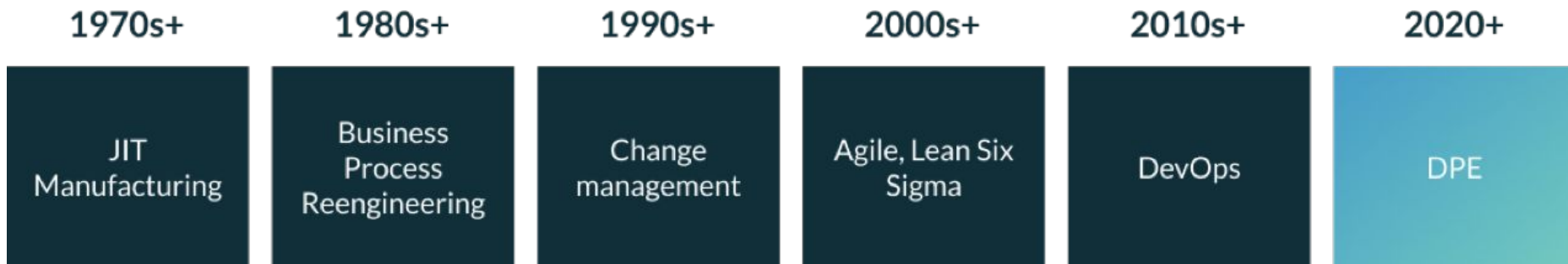


Our mission at **Gradle** is to **accelerate developer productivity** and  
**make developers happier**





What comes after DevOps?

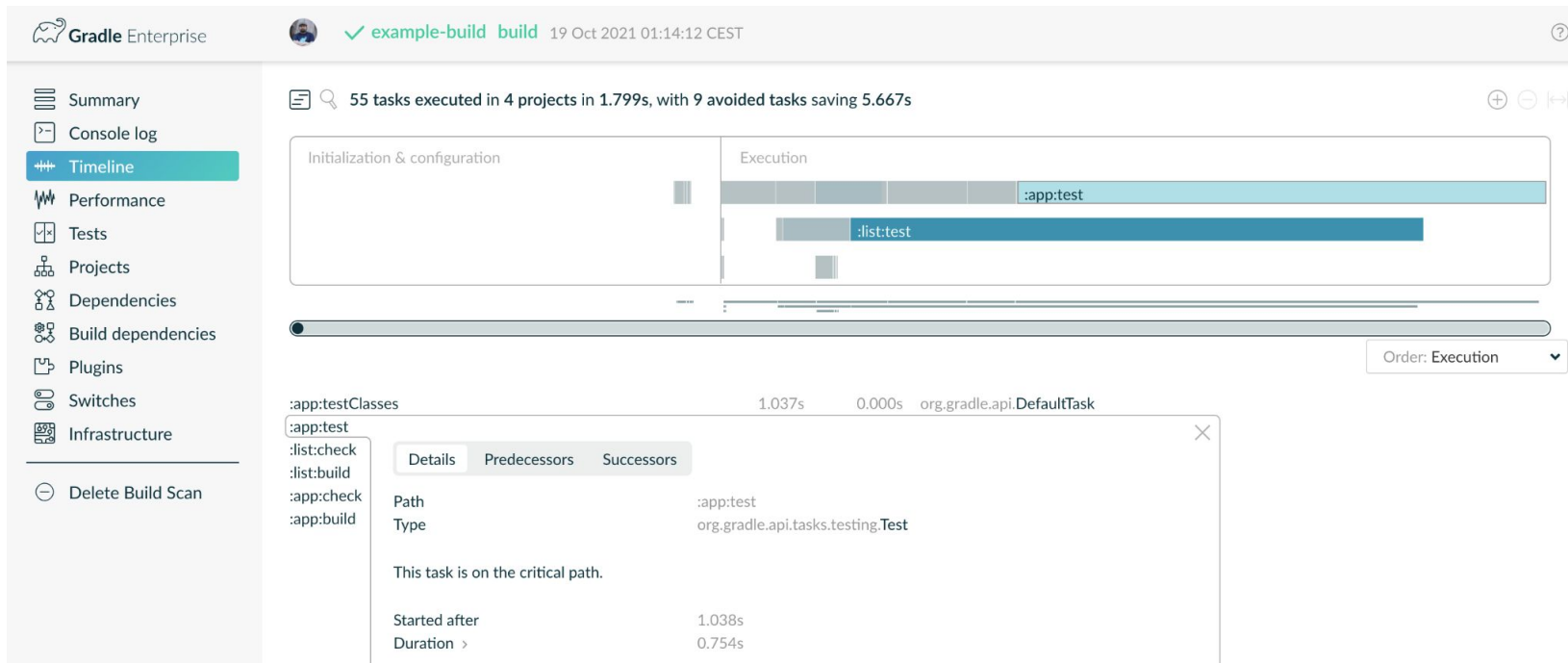


**Developer Productivity Engineering**



# “If you can’t measure it, you can’t improve it!”

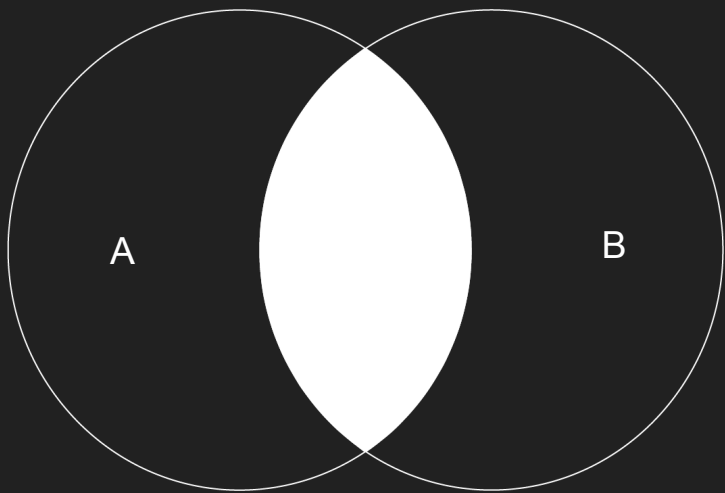
by Peter Drucker



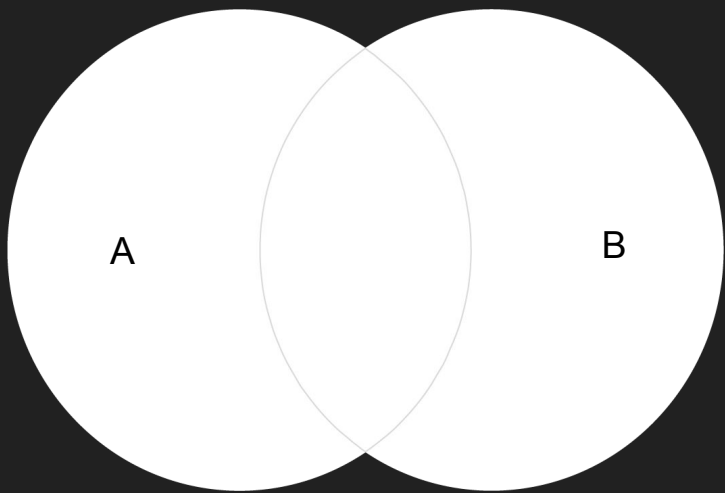




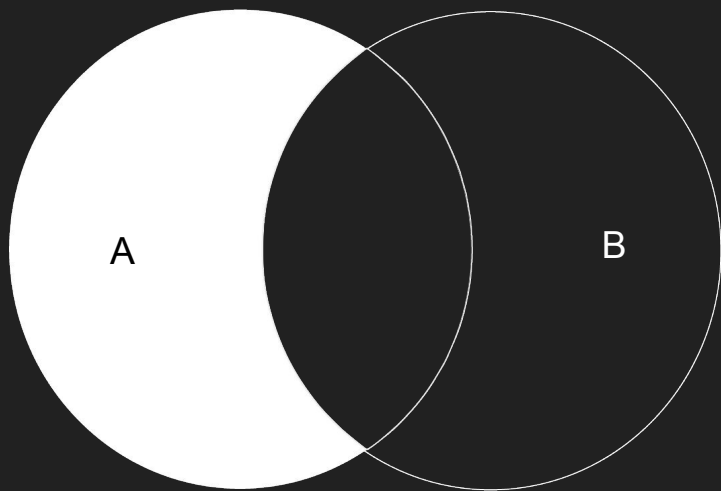




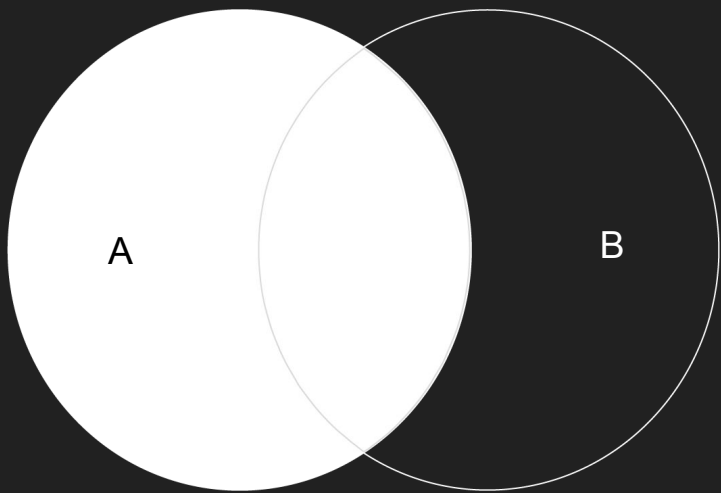
```
SELECT *  
FROM A  
      INNER JOIN B  
      ON A.ID = B.ID
```



```
SELECT *  
FROM A  
      FULL OUTER JOIN B  
      ON A.ID = B.ID
```

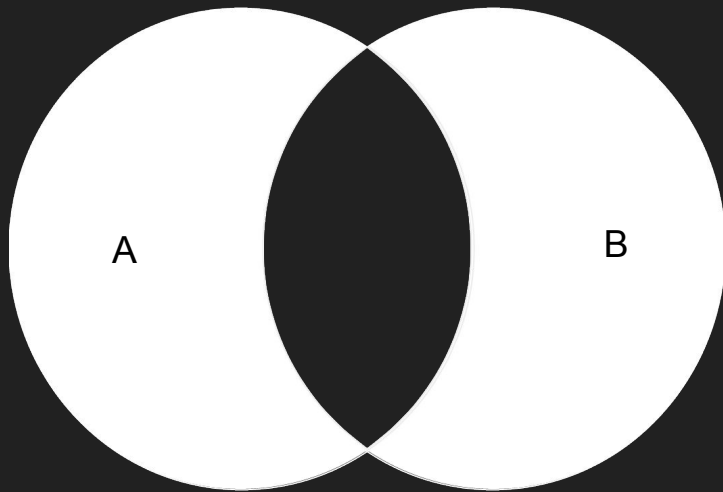


```
SELECT *  
FROM A  
      LEFT JOIN B  
      ON A.ID = B.ID
```



```
SELECT *  
FROM A  
      LEFT OUTER JOIN B  
      ON A.ID = B.ID
```

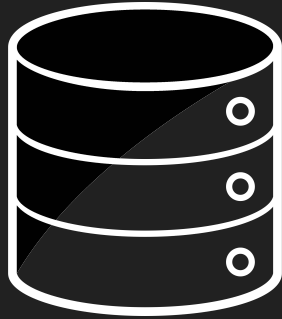




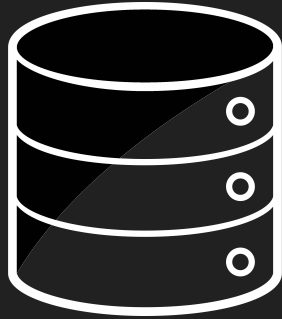
```
SELECT *  
FROM A  
      FULL OUTER JOIN B  
      ON A.ID = B.ID  
WHERE A.ID IS NULL  
AND B.ID IS NULL
```

```
SELECT P.id_person,  
       P.first_name,  
       P.last_name,  
       CONVERT(varchar(30), P.birth, 104),  
       A.id_council,  
       A.id_groupe,  
       A.num_activities  
FROM person P JOIN (SELECT id_person,  
                           MIN(id_council) id_council,  
                           MIN(id_groupe) id_groupe,  
                           COUNT(*) num_activities  
                     FROM activity  
                     GROUP BY id_person  
                   ) A ON (A.id_person = P.id_person)  
WHERE P.id_person NOT IN (SELECT id_person  
                           FROM activity  
                           WHERE id_council != 5)
```





**Database**



**Data(store)**





podcast



Java 6





Java 8

# From Imperative to Declarative

```
List<String> errors = new ArrayList<>();
int errorCount = 0;
File file = new File(fileName);
String line = file.readLine();
while (errorCount < 40 && line != null) {
    if (line.startsWith("ERROR")) {
        errors.add(line);
        errorCount++;
    }
    line = file.readLine();
}
```

```
List<String> errors =
    Files.lines(Paths.get(fileName))
        .filter(l -> l.startsWith("ERROR"))
        .limit(40)
        .collect(toList())
```



## Doing some Peach Iced Tea

### Ingredients:

- 3.5L of water
  - 4 Teabags
  - 2 peaches (sliced)
  - Honey
- 

### Instructions:

1. Add tea bags and peaches slices to a large JUG
2. Pour in 1.5l boiling water over and stir
3. Leave overnight in the fridge
4. Remove tea bags and peaches and top up with the rest of the water
5. Add some honey to taste...



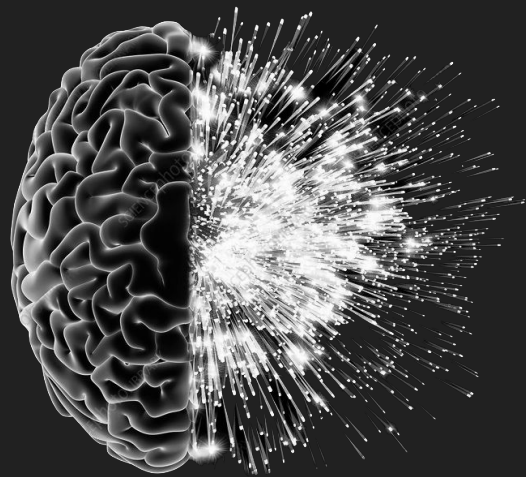
## Asking for Peach Iced Tea

“

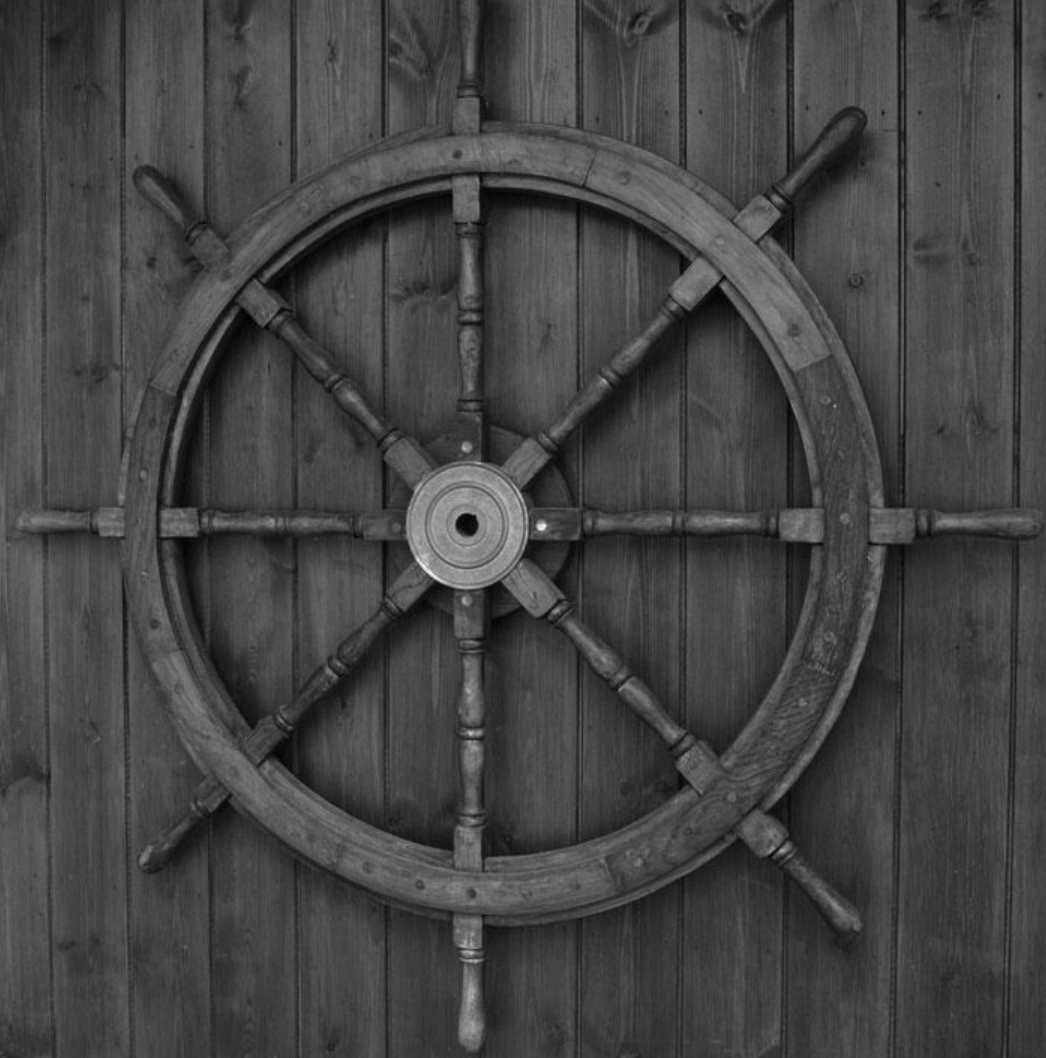
*Can I have a  
Peach iced tea*

*Please ?*

”







```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.7.9
          ports:
            - containerPort: 80
```



10 YEARS  
LATER

SQL

is just

**Declarative**

TABLE	: Stream<Tuple<..>>
SELECT	: map()
DISTINCT	: distinct()
JOIN	: flatMap()
WHERE / HAVING	: filter()
GROUP BY	: collect()
ORDER BY	: sorted()
UNION ALL	: concat()
...	



**KEEP  
CALM  
AND  
USE  
SQL**



+

SQL

How ?

First, JDBC !

```
var stmt = connection.prepareStatement("""
    SELECT TITLE, URL
    FROM PODCAST
    WHERE ID = ?
""");

// Statement are String based and should
// be constructed using only String
// methods (concat, join...)

stmt.setObject(1, id);

// Positional Parameters...

var rs = stmt.executeQuery();
rs.next();

// Managing spatial cursor position

if (rs.isAfterLast()) {
    return Optional.empty();
}

var title = rs.getString("TITLE");
var url = rs.getString("URL");

// Field name duplication
// for every one

rs.close();
stmt.close();

// Result should be closed...
// And Statement too

return Optional.of(new Podcast(id, title, url));
```



**JDBC is OLD !**



Very old (**1997**) in **Java 1.1**

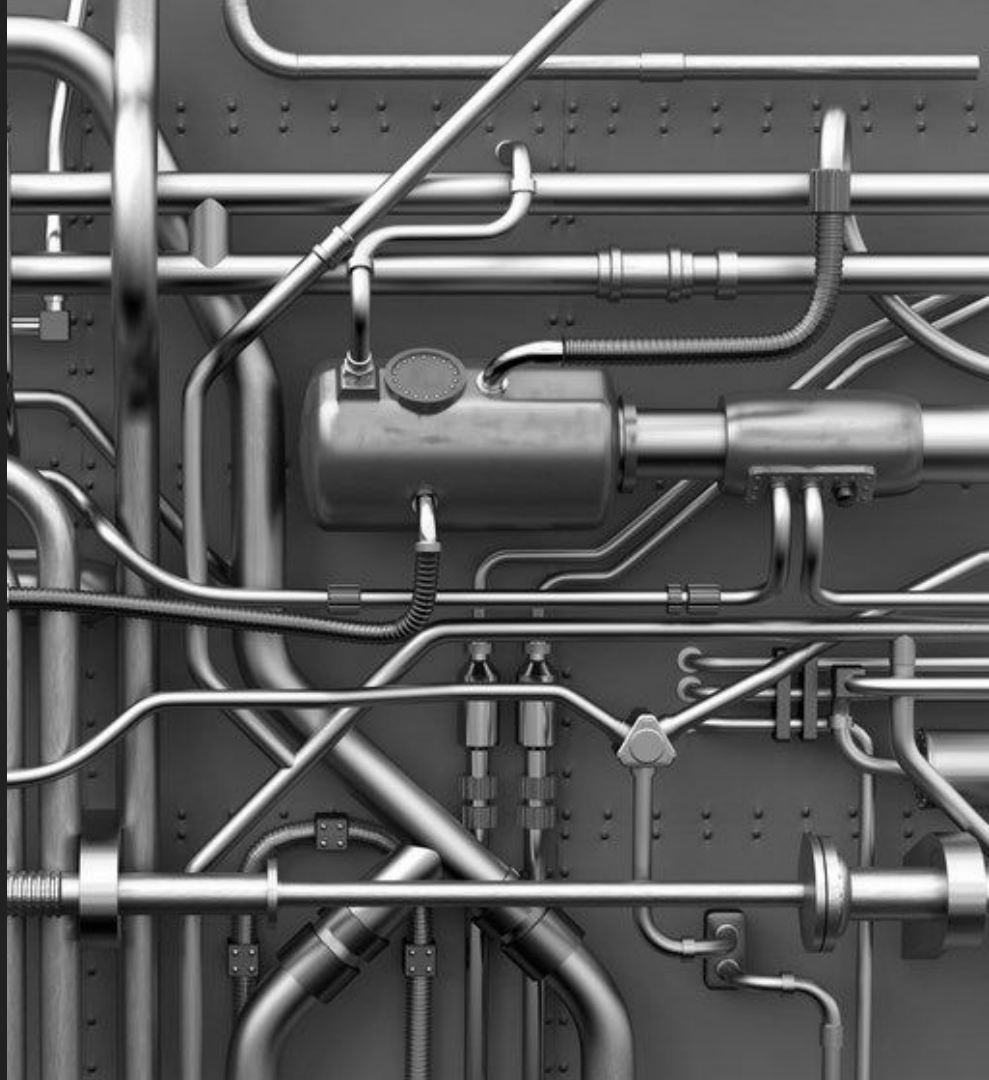


But one of the  
**most important**  
JAVA API



One of the  
**most supported**  
JAVA API

But very  
**low level**  
API



**JDBC**

**Database**



then, **EJB 2.0** Entity Beans

*Luck*

Hopefully, I've never used them...



```
public interface Podcast extends EJBObject {  
    UUID getId();  
    String getTitle();  
    String getUrl();  
  
    @Override  
    void remove();  
}
```

```
public interface PodcastRepository extends EJBHome {  
    Podcast create(UUID id);  
    Podcast find(UUID id);  
}
```

```
<enterprise-beans>
  <entity>
    <description>no description</description>
    <display-name>Podcast</display-name>
    <ejb-name>Podcast</ejb-name>
    <home>app.Podcast</home>
    <remote>app.Podcast</remote>
    <ejb-class>app.Podcast</ejb-class>
    <persistence-type>Container</persistence-type>
    <cmp-version>2.x</cmp-version>
    <abstract-schema-name>Podcast</abstract-schema-name>
    <reentrant>False</reentrant>
    <cmp-field><field-name>id</field-name></cmp-field>
    <cmp-field><field-name>title</field-name></cmp-field>
    <cmp-field><field-name>url</field-name></cmp-field>
    ...
  </entity>
</enterprise-beans>
```

```
<enterprise-beans>
  <entity>
    <query>
      <description></description>
      <query-method>
        <method-name>findAll</method-name>
        <method-params/>
      </query-method>
      <ejb-ql>SELECT OBJECT(p) FROM PODCAST p ORDER BY ID ASC</ejb-ql>
    </query>
    <query>
      <description></description>
      <query-method>
        <method-name>findOne</method-name>
        <method-params>
          <method-param>java.util.UUID</method-param>
        </method-params>
      </query-method>
      <ejb-ql>SELECT OBJECT(p) FROM PODCAST p WHERE p.id = ?1</ejb-ql>
    </query>
  </entity>
</enterprise-beans>
```

```
<ejb-q1>  
SELECT OBJECT(p)  
  FROM PODCAST p  
ORDER BY ID ASC  
</ejb-q1>
```

not SQL

**EJB Query Language**

**EJB**

**Application Server**

**JDBC**

**Database**





Rise of the  
**O**bject **R**elational **M**apping  
(aka **ORM**)





**ORM**

**JDBC**

**Database**

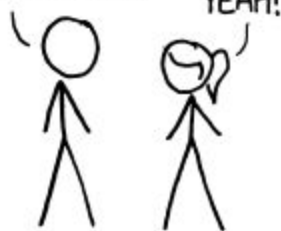


In **Java** world,  
we like to create  
**Standard**

HOW STANDARDS PROLIFERATE:  
(SEE: A/C CHARGERS, CHARACTER ENCODINGS, INSTANT MESSAGING, ETC.)

SITUATION:  
THERE ARE  
14 COMPETING  
STANDARDS.

14?! RIDICULOUS!  
WE NEED TO DEVELOP  
ONE UNIVERSAL STANDARD  
THAT COVERS EVERYONE'S  
USE CASES.



YEAH!

SOON:

SITUATION:  
THERE ARE  
15 COMPETING  
STANDARDS.

Java Persistence API  
is born

**JPA**

**ORM**

**JDBC**

**Database**



```
@Entity
@Table(
    name = "PODCAST",
    uniqueConstraints = @UniqueConstraint(columnNames={"id", "url"})
)
public class Podcast {

    @Id
    @GeneratedValue
    @Column(columnDefinition = "UUID")
    private UUID id;
    @Column(name = "TITLE")
    private String title;
    private String url;

    ...

}
```

```
@Entity
@Table(
    name = "PODCAST",
    uniqueConstraints = @UniqueConstraint(columnNames={"id", "url"})
)
public class Podcast {

    ...

    public UUID getId() { return id; }
    public void setId(UUID id) { this.id = id; }

    public String getTitle() { return title; }
    public void setTitle(String title) { this.title = title; }

    public String getUrl() { return url; }
    public void setUrl(String url) { this.url = url; }

    ...

}
```



```
@Entity
@Table(
    name = "PODCAST",
    uniqueConstraints = @UniqueConstraint(columnNames={"id", "url"})
)
public class Podcast {

    ...

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Podcast podcast = (Podcast) o;
        return Objects.equals(id, podcast.id) &&
            Objects.equals(title, podcast.title) &&
            Objects.equals(url, podcast.url);
    }

    @Override
    public int hashCode() {
        return Objects.hash(id, title, url);
    }
}
```

**Object** to database

```
public record PodcastRepository(EntityManager em) {  
  
    public List<Podcast> findAll() {  
        return em.createQuery("FROM Podcast", Podcast.class).getResultList();  
    }  
  
    public Optional<Podcast> findOneById(UUID id) {  
        CriteriaBuilder cb = em.getCriteriaBuilder();  
  
        CriteriaQuery<Podcast> cq = cb.createQuery(Podcast.class);  
        Root<Podcast> root = cq.from(Podcast.class);  
        cq.select(root);  
  
        cq.where(cb.equal(root.get("id"), id));  
  
        var podcast = em.createQuery(cq).getSingleResult();  
  
        return Optional.ofNullable(podcast);  
    }  
}
```

JPQL, not SQL, again...

Not type safe...

```
public List<Podcast> findAll() {  
    return em.createQuery("FROM Podcast", Podcast.class).getResultList();  
}
```

**Criteria Query**, not **SQL** again...

Not type safe...

```
public Optional<Podcast> findOneById(UUID id) {
    CriteriaBuilder cb = em.getCriteriaBuilder();

    CriteriaQuery<Podcast> query = cb.createQuery(Podcast.class);
    Root<Podcast> p = query.from(Podcast.class);

    query
        .select(p)
        .where(
            cb.equal(p.get("id"), id)
        );

    var podcast = em.createQuery(query).getSingleResult();
    return Optional.ofNullable(podcast);
}
```

**SQL**-ish API

Fails only at runtime

**JPQL + Criteria Query**

**JPA**

**ORM**

**JDBC**

**Database**



And with **2** entities ?

```
@Entity
public class Podcast {

    @Id
    @GeneratedValue
    @Column(columnDefinition = "UUID")
    private UUID id;

    @OneToMany(mappedBy = "podcast")
    private List<Item> items = new ArrayList<>();

}
```

**JOIN** managed by annotations

```
@Entity
public class Item {
    @Id
    @GeneratedValue
    @Column(columnDefinition = "UUID")
    private UUID id;

    @ManyToOne(fetch = FetchType.LAZY)
    private Podcast podcast;

}
```



**Annotation Hell**  
begins 🔥

**Hard** (impossible?) to test

The diagram features a central code snippet for a JPA entity. A red line extends from the top of the snippet to the text 'Hard (impossible?) to test'. Another red line extends from the bottom of the snippet to the text 'Only triggers side effects'. A third red line extends from the right side of the snippet to the text 'Generated queries depend on object values'. The text '@ManyToMany(fetch = FetchType.LAZY)' in the code is highlighted with a red rectangular box.

```
@Entity
public class Item {

    @ManyToMany(fetch = FetchType.LAZY)
    private Podcast podcast;

}
```

**Only** triggers **side effects**

**Generated** queries  
depend on object values

All of this to replace the **JOIN** keyword in SQL...

```
@NamedEntityGraph(  
    name = "podcast-item-graph",  
    attributeNodes = {  
        @NamedAttributeNode("items")  
    },  
    subgraphs = {  
        @NamedSubgraph(  
            name = "items-tags",  
            attributeNodes = {  
                @NamedAttributeNode("tags")  
            }  
        )  
    }  
)  
  
@Entity  
@Table(name = "PODCAST")  
public class Podcast {}
```



Just... why ?

# ORM

Trying to solve problem  
we should not have  
if we **don't use it** first...

Did you ever change  
your **database** layer?

Did you ever change  
your **database** ?

And we can **dig** deeper...





Spring Data JPA




Micronaut Data



Quarkus with Panache

```
public interface PodcastRepository extends Repository<Podcast, UUID> {  
    List<Podcast> findByTitleAndUrl(String title, String url);  
}
```



Framework naming convention,  
**maybe** checked at compile time...  
or not.

```
public interface PodcastRepository extends Repository<Podcast, UUID> {  
    List<Podcast> findByTitleAndUrl(String title, String url);  
}
```

SELECT \*

FROM podcast

WHERE title = ?1

AND url = ?2;

**Framework  
Abstraction Layer**

**JPQL + Criteria Query**

**JPA**

**ORM**

**JDBC**

**Database**



# What Java ORM do you prefer, and why?

Asked 11 years, 7 months ago   Active 3 years, 9 months ago   Viewed 211k times



It's a pretty open ended question. I'll be starting out a new project and am looking at different ORMs to integrate with database access.

263



Do you have any favorites? Are there any you would advise staying clear of?



105

java

orm



I have stopped using ORMs.

239



The reason is not any great flaw in the concept. Hibernate works well. Instead, I have found that queries have low overhead and I can fit lots of complex logic into large SQL queries, and shift a lot of my processing into the database.



So consider just using the JDBC package.

share edit follow flag

answered Sep 27 '09 at 11:22



David Crawshaw

9,463 ● 6 ● 34 ● 39

# ORM

SQL  
TTノ(๖\_๖ノ)



**Framework  
Abstraction Layer**

**JPQL + Criteria Query**

**JPA**

**ORM**

**JDBC**

**Database**

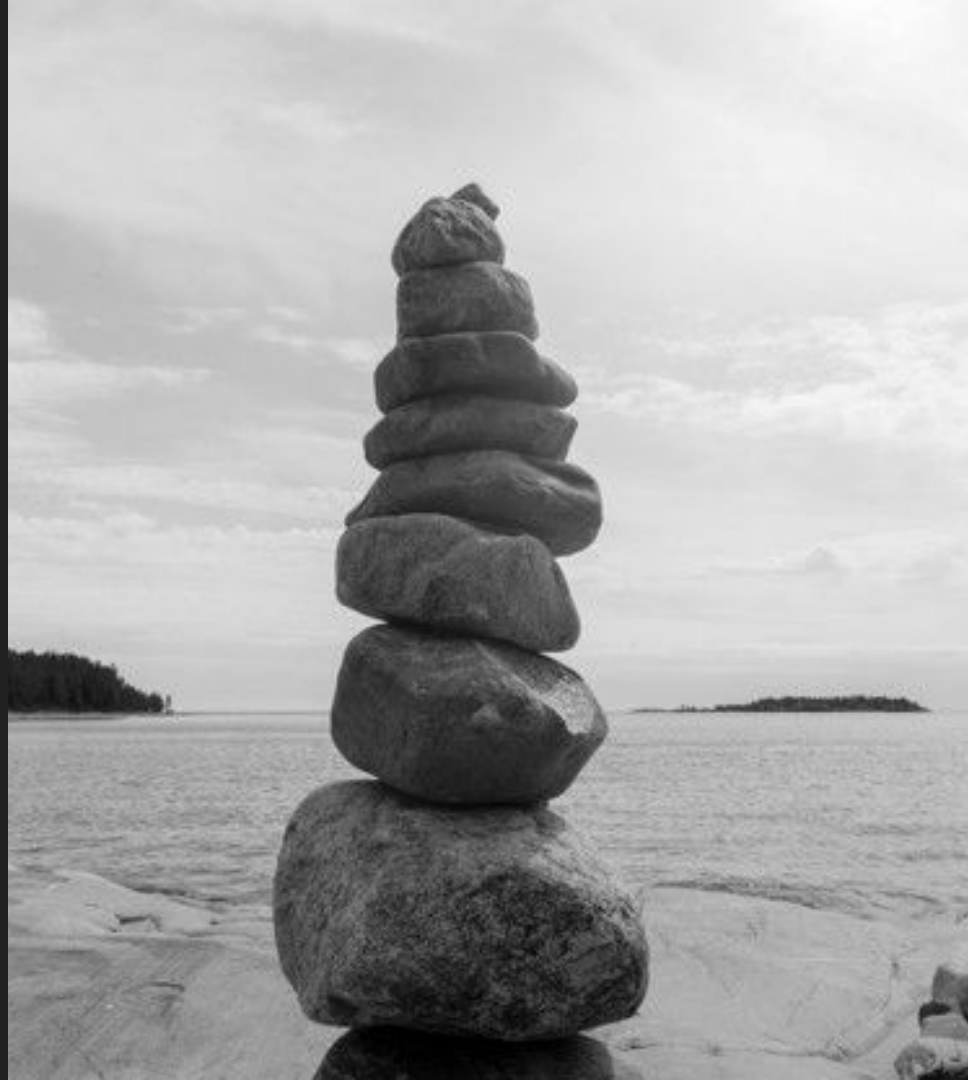


Writing SQL  
safely with



JDBC

Database





Is dual licensed



Is here to **simplify** our work...



Is a **JAVA DSL** over **SQL**

SELECT

PODCAST.ID ,  
PODCAST.TITLE ,  
PODCAST.URL

FROM

PODCAST

ORDER BY

PODCAST.ID ASC

```
query .select ( field("PODCAST.ID"),  
                field("PODCAST.TITLE"),  
                field("PODCAST.URL")      )  
      .from (   table("PODCAST")         )  
      .orderBy( field("PODCAST.ID").asc() )
```

```
public List<Podcast> findAll() {  
  
    var id = field("PODCAST.ID").cast(UUID.class);  
    var title = field("PODCAST.TITLE").cast(String.class);  
    var url = field("PODCAST.URL").cast(String.class);  
  
    return query  
        .select(id, title, url)  
        .from(table("PODCAST"))  
        .orderBy(id.asc())  
        .fetch(it -> new Podcast(it.get(id), it.get(title), it.get(url)))  
}
```

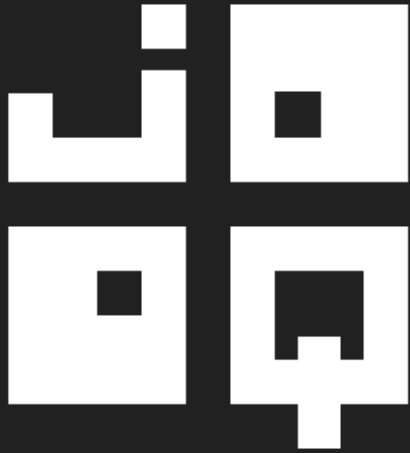


If you know **SQL**  
you know **JOOQ**

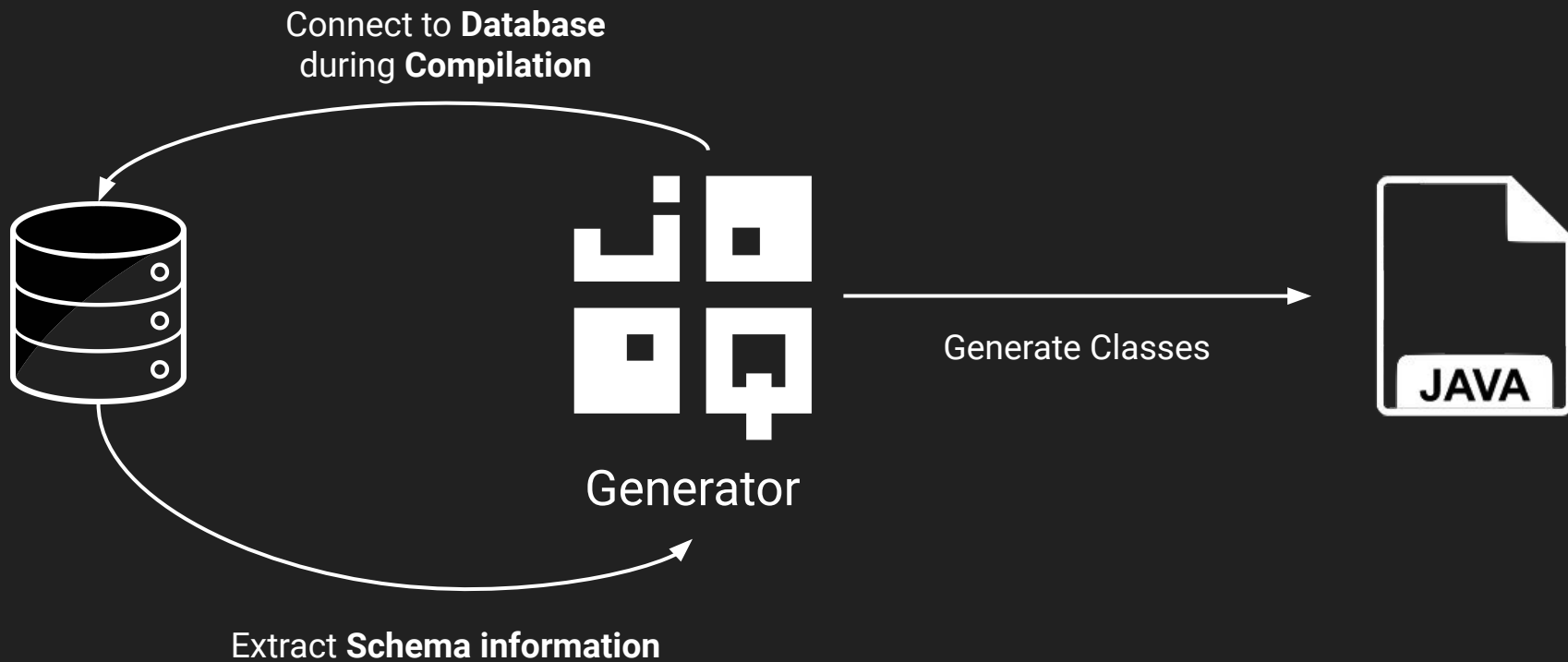
But, this is  
not **type-safe** ?



**WHAT?**

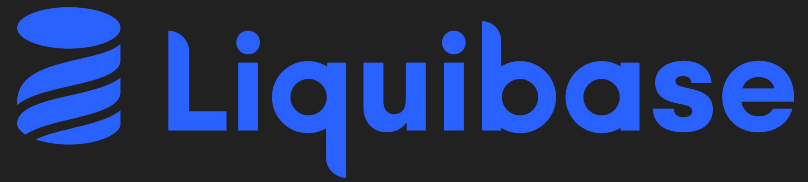


**Is Database first !**



# Tips

## Use Database versioning Tool





Java CLI



Ant



Generator



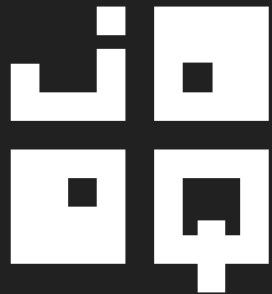
Apache Maven



Gradle



Java 7+



Generator



Scala



Kotlin

SELECT

PODCAST.ID ,  
PODCAST.TITLE ,  
PODCAST.URL

FROM

PODCAST

ORDER BY

PODCAST.ID ASC



```
query .select (          PODCAST.ID),  
                                PODCAST.TITLE),  
                                PODCAST.URL))  
    .from (                PODCAST )  
    .orderBy(  
        PODCAST.ID        .asc() )
```

```
public List<Podcast> findAll() {  
    return query  
        .select(PODCAST.ID, PODCAST.TITLE, PODCAST.URL)  
        .from(PODCAST)  
        .orderBy(PODCAST.ID.asc())  
        .fetch(it -> new Podcast(  
            it.get(PODCAST.ID),  
            it.get(PODCAST.TITLE),  
            it.get(PODCAST.URL))  
        );  
}
```

```
public Optional<Podcast> findOne(UUID id) {  
    return query  
        .select(PODCAST.ID, PODCAST.TITLE, PODCAST.URL)  
        .from(PODCAST)  
        .where(PODCAST.ID.eq(id))  
        .fetchOptional(it -> new Podcast(  
            it.get(PODCAST.ID),  
            it.get(PODCAST.TITLE),  
            it.get(PODCAST.URL))  
        );  
}
```

This is  
really **type-safe** ?



```
/**
 * This class is generated by jOOQ.
 */
public class Podcast extends TableImpl<PodcastRecord> {

    public static final Podcast PODCAST = new Podcast();

    public final TableField<PodcastRecord, UUID> ID =
        createField(
            DSL.name("id"),
            org.jooq.impl.SQLDataType.UUID.nullable(false),
            this,
            ""
        );
}
```



Type information

How much  
**type-safe**  
it is ?



```
public Optional<Podcast> findOne(UUID id) {  
    return query  
        .select(PODCAST.ID, PODCAST.TITLE, PODCAST.URL)  
        .from(PODCAST)  
        .where(PODCAST.ID.eq(id))  
        .fetchOptional(it -> new Podcast(  
            it.get(PODCAST.ID),    // UUID  
            it.get(PODCAST.TITLE), // String  
            it.get(PODCAST.URL))   // URI  
        );  
}
```

```
public Optional<Podcast> findOne(UUID id) {  
    return query  
        .select(PODCAST.ID, PODCAST.TITLE, PODCAST.URL)  
        .from(PODCAST)  
        .where(PODCAST.ID.eq("05863ea0-1bd7-4d4a-9c72-283fdfe4a393"))  
        .fetchOptional(it -> new Podcast(  
            it.get(PODCAST.ID),  
            it.get(PODCAST.TITLE),  
            it.get(PODCAST.URL))  
        );  
}
```



**Errors** will be  
detected **during**  
**compilation**

What is  
the generated  
**SQL Query** ?

```
public Optional<Podcast> findOne(UUID id) {  
    return query  
        .select(PODCAST.ID, PODCAST.TITLE, PODCAST.URL)  
        .from(PODCAST)  
        .where(PODCAST.ID.eq(id))  
        .fetchOptional(it -> new Podcast(  
            it.get(PODCAST.ID),  
            it.get(PODCAST.TITLE),  
            it.get(PODCAST.URL))  
        );  
}
```

```
select
  "podcast"."id",
  "podcast"."title",
  "podcast"."url"
from "podcast"
where "podcast"."id" = cast(? as uuid);
```

What are  
**query parameters ?**

```
select
  "podcast"."id",
  "podcast"."title",
  "podcast"."url"
from "podcast"
where "podcast"."id" = 'b08c7404-d9de-43d0-b1a3-25b8a26fa67d' ;
```

And you can **see result** too  
directly from your **console logs** !

```
<dependency>  
  <groupId>org.slf4j</groupId>  
  <artifactId>slf4j-api</artifactId>  
</dependency>
```

...

```
<dependency>  
  <groupId>ch.qos.logback</groupId>  
  <artifactId>logback-classic</artifactId>  
</dependency>
```



```
14:50:30.878 [main] DEBUG org.jooq.tools.LoggerListener - Executing query :
select
  "podcast"."id",
  "podcast"."title",
  "podcast"."url"
from "podcast"
where "podcast"."id" = cast(? as uuid)
14:50:30.879 [main] DEBUG org.jooq.tools.LoggerListener - -> with bind values :
select
  "podcast"."id",
  "podcast"."title",
  "podcast"."url"
from "podcast"
where "podcast"."id" = 'b08c7404-d9de-43d0-b1a3-25b8a26fa67d'
14:50:30.945 [main] DEBUG org.jooq.tools.LoggerListener - Fetched result
14:50:30.946 [main] DEBUG org.jooq.tools.LoggerListener -
14:50:30.946 [main] DEBUG org.jooq.tools.LoggerListener -
14:50:30.946 [main] DEBUG org.jooq.tools.LoggerListener -
14:50:30.946 [main] DEBUG org.jooq.tools.LoggerListener -
14:50:30.946 [main] DEBUG org.jooq.tools.LoggerListener -
14:50:30.946 [main] DEBUG org.jooq.tools.LoggerListener - Fetched row(s)
```

id			title			url		
b08c7404-d9de-43d0-b1a3-25b8a26fa67d			Les Cast Codeurs			http://lescastcodeurs.libsyn.com/rss		
1								

And with **2** tables ?

```

public List<PodcastWithCover> findThreeWithCover() {
    return query
        .select( PODCAST.ID, PODCAST.TITLE, PODCAST.URL,
                  COVER.URL, COVER.HEIGHT, COVER.WIDTH)
        .from(PODCAST)
            .innerJoin(COVER).on(PODCAST.COVER_ID.eq(COVER.ID))
        .orderBy(PODCAST.ID.asc())
        .limit(3)
        .fetch(it -> new PodcastWithCover(
            it.get(PODCAST.ID),
            it.get(PODCAST.TITLE),
            it.get(PODCAST.URL),
            new PodcastWithCover.Cover(
                it.get(COVER.URL),
                it.get(COVER.HEIGHT),
                it.get(COVER.WIDTH)
            )
        ));
}

```

```
// Fetch only required data
select( PODCAST.ID, PODCAST.TITLE, PODCAST.URL,
        COVER.URL, COVER.HEIGHT, COVER.WIDTH)

// Type Safe join operation between tables
.from(PODCAST)
    .innerJoin(COVER)
        .on(PODCAST.COVER_ID.eq(COVER.ID))

// Standard SQL operations, nothing fancy here...
.orderBy(PODCAST.ID.asc())
.limit(3)
```

```
public List<PodcastWithCover> findThreeWithCover() {
    return query
        .select(PODCAST.ID, PODCAST.TITLE, PODCAST.URL,
                COVER.URL, COVER.HEIGHT, COVER.WIDTH)
        .from(PODCAST)
            .innerJoin(COVER).on(PODCAST.COVER_ID.eq(COVER.ID))
        .orderBy(PODCAST.ID.asc())
        .limit(3)
        .fetch(it -> new PodcastWithCover(
            it.get(PODCAST.ID),
            it.get(PODCAST.TITLE),
            it.get(PODCAST.URL),
            new PodcastWithCover.Cover(
                it.get(COVER.URL),
                it.get(COVER.HEIGHT),
                it.get(COVER.WIDTH)
            )
        ));
}
```

This is **almost**  
plain old  
**standard SQL**



***BORING***

YES.



But, it can be **sexy** too!

```
// Fetch only required data
select( PODCAST.ID, PODCAST.TITLE, PODCAST.URL,
        PODCAST.cover().URL, PODCAST.cover().HEIGHT, PODCAST.cover().WIDTH)

// Join statement with COVER automatically done by JOOQ
.from(PODCAST)

// Standard SQL operations, nothing fancy here...
.orderBy(PODCAST.ID.asc())
.limit(3)
```

And with **m-n** relationship ?

```
public List<Podcast> findAllWithItems() {  
    return query  
        .select(  
            PODCAST.ID, PODCAST.TITLE, PODCAST.URL,  
            multiset(  
                select(ITEM.ID, ITEM.TITLE, ITEM.URL)  
                    .from(ITEM)  
                    .where(ITEM.PODCAST_ID.eq(PODCAST.ID))  
            )  
            .as("items")  
            .convertFrom(it -> it.map(mapping(Podcast.Item::new)))  
        )  
        .from(PODCAST)  
        .fetch(mapping(Podcast::new));  
}
```

Type-safe mapping into a List<Item>

Mapping into Podcast

```
public record Podcast(UUID id, String title, String url, Collection<Item> items) {  
    public record Item(UUID id, String title, String url) {}  
}
```

```
select
  podcast.id,
  podcast.title,
  podcast.url,
  (
    select coalesce(
      jsonb_agg(jsonb_build_array("v0", "v1", "v2")),
      jsonb_build_array()
    )
  )
from (
  select id as "v0", title as "v1", url as "v2"
  from item
  where item.podcast_id = podcast.id
) as t
) as items
from podcast;
```

But, what if  
my **database doesn't** support  
**XXX SQL syntax** ?

```

public List<PodcastWithCover> findThreeWithCover() {
    return query
        .select( PODCAST.ID, PODCAST.TITLE, PODCAST.URL,
                  COVER.URL, COVER.HEIGHT, COVER.WIDTH)
        .from(PODCAST)
            .innerJoin(COVER).on(PODCAST.COVER_ID.eq(COVER.ID))
        .orderBy(PODCAST.ID.asc())
        .limit(3) -- LIMIT syntax specific to some Engine...
        .fetch(it -> new PodcastWithCover(
            it.get(PODCAST.ID),
            it.get(PODCAST.TITLE),
            it.get(PODCAST.URL),
            new PodcastWithCover.Cover(
                it.get(COVER.URL),
                it.get(COVER.HEIGHT),
                it.get(COVER.WIDTH)
            )
        ));
}

```

```
select
  "podcast"."id", "podcast"."title", "podcast"."url",
  "cover"."url", "cover"."height", "cover"."width"
from "podcast"
  join "cover"
    on "podcast"."cover_id" = "cover"."id"
order by "podcast"."id" asc
limit 3; -- LIMIT syntax for PostgreSQL, MySQL, MariaDB...
```



```
select top 3 -- TOP syntax for SQL Server
    "podcast"."id", "podcast"."title", "podcast"."url",
    "cover"."url", "cover"."height", "cover"."width"
from "podcast"
    join "cover"
        on "podcast"."cover_id" = "cover"."id"
order by "podcast"."id" asc;
```

```
select "v0" "id", "v1" "title", "v2" "url", "v3" "url", "v4" "height", "v5" "width"
from (
  select "x"."v0", "x"."v1", "x"."v2", "x"."v3", "x"."v4", "x"."v5", rownum "rn"
  from (
    select "podcast"."id" "v0", "podcast"."title" "v1", "podcast"."url" "v2", "cover"."url"
    "v3", "cover"."height" "v4", "cover"."width" "v5"
    from "podcast"
    join "cover"
    on "podcast"."cover_id" = "cover"."id"
    order by "v0" asc
  ) "x"
  where rownum <= 3 -- LIMIT emulated for Oracle runtime
)
where "rn" > 0 -- which is very complex to write as a developer 😊
order by "rn"
```



**converts**

**standard SQL**

to your RDBMS syntax



emulates

**non-standard SQL**

in your RDBMS syntax

```
public Podcast create(Podcast p) {  
    var id = UUID.randomUUID();  
    query  
        .insertInto(PODCAST,  
                    PODCAST.ID, PODCAST.TITLE, PODCAST.URL)  
        .values(id, p.title(), p.url())  
        .execute();  
  
    return new Podcast(id, p.title(), p.url());  
}
```



```
public Podcast create(Podcast p) {  
    var id = UUID.randomUUID();  
  
    query  
        .insertInto(PODCAST)  
        .set(PODCAST.ID, id)           // SET syntax, UPDATE-like,  
        .set(PODCAST.TITLE, p.title()) // simpler to read  
        .set(PODCAST.URL, p.url())     // available only for some  
        .execute();                   // RDBMS like MySQL  
  
    return new Podcast(id, p.title(), p.url());  
}
```







**emulates**

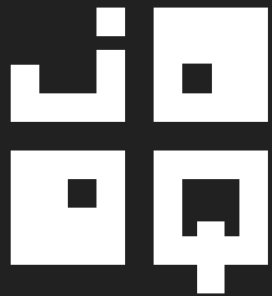
**non-standard SQL**

to simplify our life...

```
public Podcast create(Podcast p) {  
    var id = p.id() != null ? p.id() : UUID.randomUUID();  
  
    query  
        .insertInto(PODCAST)  
        .set(PODCAST.ID, id)  
        .set(PODCAST.TITLE, p.title())  
        .set(PODCAST.URL, p.url())  
        .onDuplicateKeyUpdate()           // if ID already exists,  
        .set(PODCAST.LAST_UPDATE, now()) // update instead of insert  
        .execute();                       // UPSERT style...  
  
    return new Podcast(id, p.title(), p.url());  
}
```

```
public Podcast create(Podcast p) {  
    var id = UUID.randomUUID();  
  
    query  
        .insertInto(PODCAST)  
        .set(PODCAST.ID, id)  
        .set(PODCAST.TITLE, p.title())  
        .set(PODCAST.URL, p.url())  
        .onConflict(PODCAST.URL).doUpdate() // Triggers update on  
        .set(PODCAST.LAST_UPDATE, now())    // specific conflicts  
        .execute();  
  
    return new Podcast(id, p.title(), p.url());  
}
```

```
insert into "podcast" (  
    "id",  
    "title",  
    "url"  
)  
values (  
    '9eff8121-94fe-47b0-b354-02f37f472b39',  
    'KubeCon and CloudNativeCon Europe 2020',  
    'https://www.youtube.com/playlist?list=PLj6h78yzYM201wlsM-Ma-RYhfT5LKq0XC'  
)  
on conflict ("url") do update  
set "last_update" =  
    timestamp with time zone '2020-09-09 15:08:12.328091+00:00';  
  
-- Or emulates equivalent syntax if possible...
```



becomes our  
**SQL**

*Best Friend*

But,  
***XYZ*** is **simpler**  
for my  
**CRUD**

Yes,  
but it's **not** really  
**safe**





provides

**alternatives**





provides

Active **Records**

```
@Test
public void should_fetch_one() {
    /* GIVEN */
    var id = UUID.fromString("b08c7404-d9de-43d0-b1a3-25b8a26fa67d");

    /* WHEN */
    var podcast = query.newRecord(PODCAST);
    podcast.setId(id);
    podcast.refresh(); // Triggers a SELECT SQL request

    /* THEN */
    assertThat(podcast.getTitle()).isEqualTo("Les Cast Codeurs");
    assertThat(podcast.getId()).isEqualTo(id);
    assertThat(podcast.getDescription()).isNull();
}
```

```
@Test
public void should_update_too() {
    /* GIVEN */
    var id = UUID.fromString("b08c7404-d9de-43d0-b1a3-25b8a26fa67d");
    var podcast = query.newRecord(PODCAST);
    podcast.setId(id);
    podcast.refresh();

    /* WHEN */
    podcast.setDescription("Le meilleur podcast sur la JVM ");
    podcast.update(); // Triggers an UPDATE SQL request

    /* THEN */
    assertThat(podcast.getTitle()).isEqualTo("Les Cast Codeurs");
    assertThat(podcast.getDescription())
        .isEqualTo("Le meilleur podcast sur la JVM ");
}
```

```
PodcastRecord podcast = query.newRecord(PODCAST);
```

```
podcast.store();
```

```
podcast.changed();
```

```
podcast.copy();
```

```
podcast.insert();
```

```
podcast.update();
```

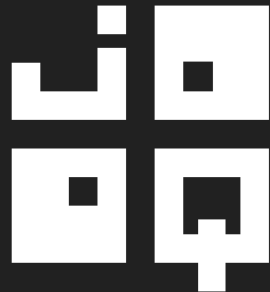
```
podcast.delete();
```

```
podcast.detach();
```

```
podcast.attach(configuration);
```

```
podcast.refresh();
```

```
// Bridge object between Database and your code, fully generated
```



provides

**DAOs**

```
<plugin>
  ...
  <configuration>
    ...
    <generator>
      <generate>
        <daos>true</daos> <!-- Activate DAOs in JOOQ Generator-->
      </generate>
    </generator>
    ...
  </configuration>
  ...
</plugin>
```

```
package com.gitlab.davinkevin.podcastserver.database.tables.pojos;
```

```
/**
```

```
 * This class is generated by jOOQ.
```

```
 */
```

```
@SuppressWarnings({ "all", "unchecked", "rawtypes" })
```

```
public class Podcast implements Serializable {
```

```
    private static final long serialVersionUID = 633654975;
```

```
    private UUID          id;
```

```
    private String        description;
```

```
    private String        title;
```

```
    public Podcast() {}
```

```
    public Podcast(Podcast value) {
```

```
        this.id = value.id;
```

```
        this.description = value.description;
```

```
        this.title = value.title;
```

```
    }
```

```
    ...  
}
```

```
package com.gitlab.davinkevin.podcastserver.database.tables.daos;

/**
 * This class is generated by jOOQ.
 */
@SuppressWarnings({ "all", "unchecked", "rawtypes" })
public class PodcastDao extends DAOImpl<PodcastRecord, Podcast, UUID> {

    public List<Podcast> fetchById(UUID... values)
    public Podcast fetchOneById(UUID value)

    public List<Podcast> fetchByDescription(String... values)
    public List<Podcast> fetchByTitle(String... values)

    public List<Podcast> fetchByUrl(String... values)
    public Podcast fetchOneByUrl(String value)

}
```



```
@Test
public void should_fetch_one() {
    /* GIVEN */
    var id = UUID.fromString("b08c7404-d9de-43d0-b1a3-25b8a26fa67d");

    /* WHEN */
    var podcast = repository.findById(id);

    /* THEN */
    assertThat(podcast.getTitle()).isEqualTo("Les Cast Codeurs");
    assertThat(podcast.getId()).isEqualTo(id);
    assertThat(podcast.getDescription()).isNull();
}
```

```
var repository = new PodcastDao(query.configuration());
```

```
repository.deleteById(id);  
repository.delete(podcast);  
repository.exists(podcast);  
repository.existsById(id);  
repository.count();  
repository.findAll();  
repository.findById();  
repository.insert(podcast);  
repository.update(podcast);
```

```
// And all methods available in ActiveRecord 🔥
```

But, I still prefer

**SQL DSL**

WHY



We can construct  
**anything**  
simply & safely

Search in ITEM  
DESCRIPTION or TITLE

Filter on TAGS (n-m relation)  
on PODCAST

Filter on ITEM.STATUS

Sorting results

Global Search

Tags

Downloaded

Publication Date

Descending

« ‹ 1 2 3 4 5 6 7 8 9 10 › »

Result pagination...

And the same request search can be scoped to a Podcast





```
public class ItemRepository(DSLContext query) {  
  
    public Page<Item> search(  
        String q, List<UUID> tagIds, List<String> statuses, PageRequest page, UUID podcastId  
    ) {  
        ...  
    }  
}
```

```
public Page<Item> search(
    String q, List<UUID> tagIds, List<String> statuses, PageRequest page, UUID podcastId
) {

    // Create condition based on q parameters on ITEM.TITLE or ITEM.DESCRPTION
    var queryCondition = q.isEmpty()
        ? noCondition()
        : ITEM.TITLE.containsIgnoreCase(q).or(ITEM.DESCRPTION.containsIgnoreCase(q));

    ...

}
```

```
public Page<Item> search(  
    String q, List<UUID> tagIds, List<String> statuses, PageRequest page, UUID podcastId  
) {  
    ...  
  
    // Create a condition based on a sub-query and IN operator  
    var tagsCondition = tagIds  
        .stream()  
        .map(it -> value(it).in(  
            select(PODCAST_TAGS.TAGS_ID)  
            .from(PODCAST_TAGS)  
            .where(ITEM.PODCAST_ID.eq(PODCAST_TAGS.PODCASTS_ID)))  
        ) // Create a condition for each tag id  
        // Accumulate those conditions into one Condition  
        .reduce(noCondition(), DSL::and);  
    ...  
  
}
```

```
public Page<Item> search(
    String q, List<UUID> tagIds, List<String> statuses, PageRequest page, UUID podcastId
) {

    ...

    // Create a condition on ITEM.STATUS
    var statusesCondition = statuses.isEmpty()
        ? noCondition()
        : ITEM.STATUS.in(statuses);

    // Create a condition on ITEM.PODCAST_ID
    var podcastCondition = podcastId == null
        ? noCondition()
        : ITEM.PODCAST_ID.eq(podcastId);

    ...

}
```

```
public Page<Item> search(
    String q, List<UUID> tagIds, List<String> statuses, PageRequest page, UUID podcastId
) {
    ...

    // Merge all those conditions with AND keyword...
    var conditions = and(queryCondition, tagsCondition, statusesCondition, podcastCondition);

    ...
}
```

```

public Page<Item> search(
    String q, List<UUID> tagIds, List<String> statuses, PageRequest page, UUID podcastId
) {
    ...
    // Creation of a COMMON TABLE EXPRESSION (CTE) fetching only required ITEMS
    var fi = name("FILTERED_ITEMS").as(
        select(
            ITEM.ID, ITEM.TITLE, ITEM.URL,
            ITEM.PUB_DATE, ITEM.DOWNLOAD_DATE, ITEM.CREATION_DATE,
            ITEM.DESCRPTION, ITEM.MIME_TYPE, ITEM.LENGTH, ITEM.FILE_NAME, ITEM.STATUS,

            ITEM.PODCAST_ID, ITEM.COVER_ID
        )
        .from(ITEM)
        .where(conditions) // Usage of our previously created conditions
        .orderBy(toOrderBy(page.sort()), ITEM.ID.asc())
        .limit(page.size() * page.page(), page.size().intValue())
    );
    ...
}

```

```

public Page<Item> search(
    String q, List<UUID> tagIds, List<String> statuses, PageRequest page, UUID podcastId
) {
    ...
    // Global query doing the global work, joining with COVER and PODCAST
    var content = query
        .with(fi) // Usage of our CTE created just before...
        .select(
            fi.field(ITEM.ID), fi.field(ITEM.TITLE), fi.field(ITEM.URL),
            fi.field(ITEM.PUB_DATE), fi.field(ITEM.DOWNLOAD_DATE), fi.field(ITEM.CREATION_DATE),
            fi.field(ITEM.DESCRPTION), fi.field(ITEM.MIME_TYPE), fi.field(ITEM.LENGTH),
            fi.field(ITEM.FILE_NAME), fi.field(ITEM.STATUS),

            PODCAST.ID, PODCAST.TITLE, PODCAST.URL,
            COVER.ID, COVER.URL, COVER.WIDTH, COVER.HEIGHT
        )
        .from(
            fi
                .innerJoin(COVER).on(fi.field(ITEM.COVER_ID).eq(COVER.ID))
                .innerJoin(PODCAST).on(fi.field(ITEM.PODCAST_ID).eq(PODCAST.ID))
        )
        .orderBy(toOrderBy(page.sort(), fi), fi.field(ITEM.ID))
        .fetch( r -> createItem(r) );

    ...
}

```

```
public Page<Item> search(
    String q, List<UUID> tagIds, List<String> statuses, PageRequest page, UUID podcastId
) {
    ...

    // Fetching metadata for pagination to fulfill Page<T>
    var totalElements = query
        .select(countDistinct(ITEM.ID))
        .from(ITEM)
        .where(conditions) // Reuse of our previously created conditions
        .fetchOne(countDistinct(ITEM.ID));

    return Page.of(
        content,
        totalElements,
        page
    );
}
```



```

with FILTERED_ITEMS as (
  select
    item.id,item.title,item.url,
    item.pub_date,item.download_date,item.creation_date,
    item.description,item.mime_type,item.length,item.file_name,item.status,
    item.podcast_id,item.cover_id
  from item
  where (
    '21fada5d-73ae-419a-8750-0074769e26d9' in (
      select podcast_tags.tags_id from podcast_tags where item.podcast_id = podcast_tags.podcasts_id
    )
    and item.status in ('FINISH')
  )
  order by item.pub_date desc, item.id asc
  limit 5 offset 0
)
select
  FILTERED_ITEMS.id,FILTERED_ITEMS.title,FILTERED_ITEMS.url,
  FILTERED_ITEMS.pub_date,FILTERED_ITEMS.download_date,FILTERED_ITEMS.creation_date,
  FILTERED_ITEMS.description,FILTERED_ITEMS.mime_type,FILTERED_ITEMS.length,
  FILTERED_ITEMS.file_name,FILTERED_ITEMS.status,
  podcast.id,podcast.title,podcast.url,
  cover.id,cover.url,cover.width,cover.height
from FILTERED_ITEMS
join cover
  on FILTERED_ITEMS.cover_id = cover.id
join podcast
  on FILTERED_ITEMS.podcast_id = podcast.id
order by
  FILTERED_ITEMS.pub_date desc, FILTERED_ITEMS.id

```

**Only 1 optimized request to  
fetch our data**

We are transforming

**input** → **SQL**

With all  
**SQL capabilities**

Because mostly known  
**SQL features**  
are from SQL-92

Since 92, a lot of  
**evolutions**  
happen in **SQL world**

We are using  
**Common Table Expression**  
(specified in **SQL-99**)  
to simplify our query

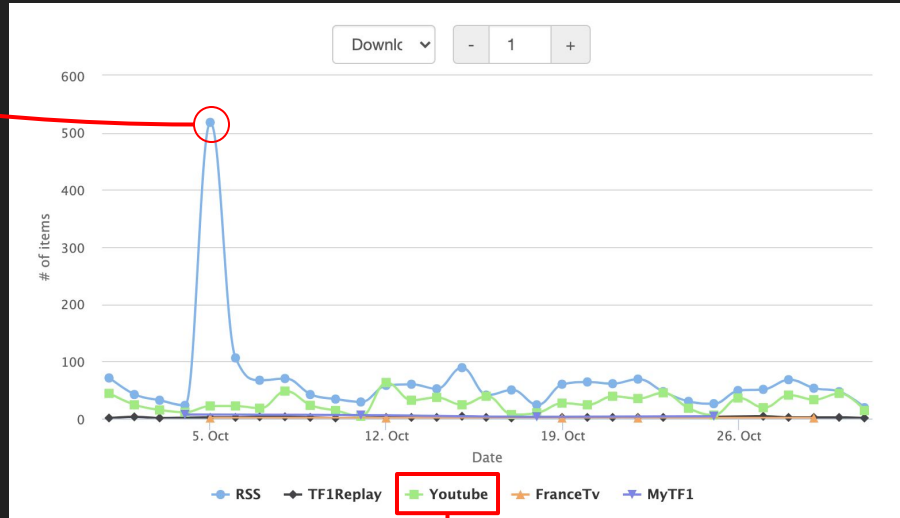
```
with FILTERED_ITEMS as (  
  select  
    item.id,item.title,item.url,  
    item.pub_date,item.download_date,item.creation_date,  
    item.description,item.mime_type,item.length,item.file_name,item.status,  
    item.podcast_id,item.cover_id  
  from item  
  where (  
    '21fada5d-73ae-419a-8750-0074769e26d9' in (  
      select podcast_tags.tags_id from podcast_tags where item.podcast_id = podcast_tags.podcasts_id  
    )  
    and item.status in ('FINISH')  
  )  
  order by item.pub_date desc, item.id asc  
  limit 5 offset 0  
)  
select  
  FILTERED_ITEMS.id,FILTERED_ITEMS.title,FILTERED_ITEMS.url,  
  FILTERED_ITEMS.pub_date,FILTERED_ITEMS.download_date,FILTERED_ITEMS.creation_date,  
  FILTERED_ITEMS.description,FILTERED_ITEMS.mime_type,FILTERED_ITEMS.length,  
  FILTERED_ITEMS.file_name,FILTERED_ITEMS.status,  
  podcast.id,podcast.title,podcast.url,  
  cover.id,cover.url,cover.width,cover.height  
from FILTERED_ITEMS  
join cover  
  on FILTERED_ITEMS.cover_id = cover.id  
join podcast  
  on FILTERED_ITEMS.podcast_id = podcast.id  
order by  
  FILTERED_ITEMS.pub_date desc, FILTERED_ITEMS.id
```

---

Using CTE instead of Nested Queries

# Statistics on download

NumberOfItemByDate



StatsPodcastByType



```

public List<StatsPodcastByType> findStatByTypeAndPubDate(OffsetDateTime startingFrom) {
    var date = trunc(ITEM.PUB_DATE);

    return query
        .select(
            PODCAST.TYPE,
            date,
            count()
        )
        .from(ITEM.innerJoin(PODCAST).on(ITEM.PODCAST_ID.eq(PODCAST.ID)))
        .where(ITEM.PUB_DATE.isNotNull())
        .and(
            date.gt(offsetDateTime(startingFrom).minus(10))
        )
        .groupBy(PODCAST.TYPE, date)
        .orderBy(PODCAST.TYPE, date)
        // Result<Record3<PODCAST.TYPE, OffsetDateTime, count()>>
        // [
        //     Tuple("Youtube", 2020-11-17, 10),
        //     Tuple("Youtube", 2020-11-16, 15),
        //     Tuple("RSS", 2020-11-17, 15),
        //     ...
        // ]
        .fetch(it -> convertToStats(it));
}

```

```
select
  podcast.type,
  date_trunc('day', item.pub_date),
  count(*)
from item
  join podcast
    on item.podcast_id = podcast.id
where (
  item.pub_date is not null
  and date_trunc('day', item.pub_date) > (timestamp with time zone '2020-07-01
00:00:00+00:00' + -(10) * interval '1 day')
)
group by podcast.type, date_trunc('day', item.pub_date)
order by podcast.type, date_trunc('day', item.pub_date)
```

podcast.type	date_trunc('day', item.pub_date)	count(*)
'RSS'	'2020-06-22'	2
'RSS'	'2020-06-23'	3
'RSS'	'2020-06-24'	1
'Youtube'	'2020-06-22'	9
'Youtube'	'2020-06-23'	1
'Youtube'	'2020-06-24'	16

What if I want  
**aggregated / cumulated**  
values ?

**Don't** do it  
in

**memory !**

Use

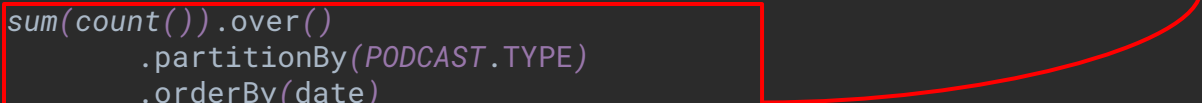
# OVER

SQL feature

(specified in **SQL-2003**)

```
public List<StatsPodcastByType> _findStatByTypeAndPubDateCumulated(OffsetDateTime startingFrom) {
    var date = trunc(ITEM.PUB_DATE);
    return query
        .select(
            PODCAST.TYPE,
            date,
            sum(count()).over(
                .partitionBy(PODCAST.TYPE)
                .orderBy(date)
                .rowsBetweenUnboundedPreceding()
                .andCurrentRow()
            )
        )
        .from(ITEM.innerJoin(PODCAST).on(ITEM.PODCAST_ID.eq(PODCAST.ID)))
        .where(ITEM.PUB_DATE.isNotNull())
        .and(
            date.gt(offsetDateTime(startingFrom).minus(10))
        )
        .groupBy(PODCAST.TYPE, date)
        .orderBy(PODCAST.TYPE, date)
        // Record<PODCAST.TYPE, count(), OffsetDateTime>
        .fetch(it -> convertToStats(it));
}
```

Partition by *PODCAST.TYPE*



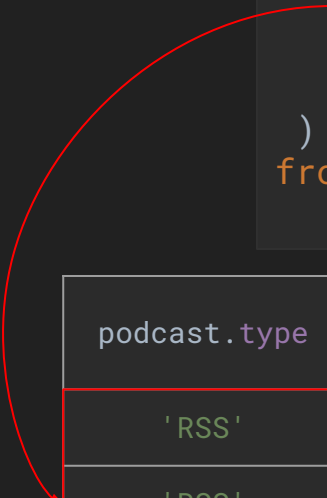
```
select
  podcast.type,
  date_trunc('day', item.pub_date),
  sum(count(*)) over (
    partition by podcast.type
    order by date_trunc('day', item.pub_date)
    rows between unbounded preceding and current row
  )
from item
join podcast
  on item.podcast_id = podcast.id
where (
  item.pub_date is not null
  and date_trunc('day', item.pub_date) > (timestamp with time zone '2020-07-01
00:00:00+00:00' + -(10) * interval '1 day')
)
group by podcast.type, date_trunc('day', item.pub_date)
order by podcast.type, date_trunc('day', item.pub_date)
```

Partition by PODCAST.TYPE





```
select
  podcast.type,
  date_trunc('day', item.pub_date),
  sum(count(*)) over (
    partition by podcast.type
    order by date_trunc('day', item.pub_date)
    rows between unbounded preceding and current row
  )
from ...
```

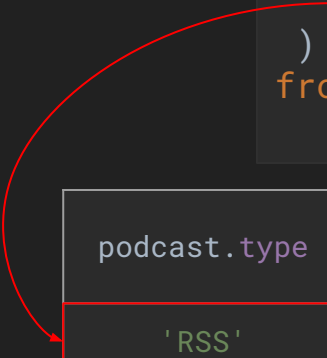


podcast.type	date_trunc('day', item.pub_date)	count(*)	sum(count(*)) over
'RSS'	'2020-06-22'	2	
'RSS'	'2020-06-23'	3	
'RSS'	'2020-06-24'	1	
'Youtube'	'2020-06-22'	9	
'Youtube'	'2020-06-23'	1	
'Youtube'	'2020-06-24'	16	

```
select
  podcast.type,
  date_trunc('day', item.pub_date),
  sum(count(*)) over (
    partition by podcast.type
    order by date_trunc('day', item.pub_date)
    rows between unbounded preceding and current row
  )
from ...
```

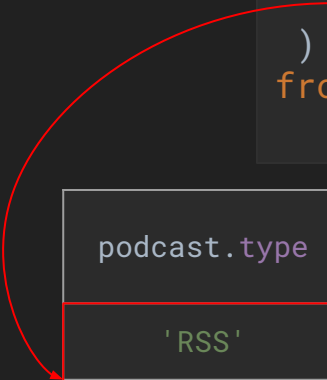
podcast.type	date_trunc('day', item.pub_date)	count(*)	sum(count(*)) over
'RSS'	'2020-06-22'	2	
'RSS'	'2020-06-23'	3	
'RSS'	'2020-06-24'	1	
'Youtube'	'2020-06-22'	9	
'Youtube'	'2020-06-23'	1	
'Youtube'	'2020-06-24'	16	

```
select
  podcast.type,
  date_trunc('day', item.pub_date),
  sum(count(*)) over (
    partition by podcast.type
    order by date_trunc('day', item.pub_date)
    rows between unbounded preceding and current row
  )
from ...
```




podcast.type	date_trunc('day', item.pub_date)	count(*)	sum(count(*)) over
'RSS'	'2020-06-22'	2	2
'RSS'	'2020-06-23'	3	
'RSS'	'2020-06-24'	1	
'Youtube'	'2020-06-22'	9	
'Youtube'	'2020-06-23'	1	
'Youtube'	'2020-06-24'	16	

```
select
  podcast.type,
  date_trunc('day', item.pub_date),
  sum(count(*)) over (
    partition by podcast.type
    order by date_trunc('day', item.pub_date)
    rows between unbounded preceding and current row
  )
from ...
```



podcast.type	date_trunc('day', item.pub_date)	count(*)	sum(count(*)) over
'RSS'	'2020-06-22'	2	2
'RSS'	'2020-06-23'	3	5
'RSS'	'2020-06-24'	1	
'Youtube'	'2020-06-22'	9	
'Youtube'	'2020-06-23'	1	
'Youtube'	'2020-06-24'	16	

```
select
  podcast.type,
  date_trunc('day', item.pub_date),
  sum(count(*)) over (
    partition by podcast.type
    order by date_trunc('day', item.pub_date)
    rows between unbounded preceding and current row
  )
from ...
```



podcast.type	date_trunc('day', item.pub_date)	count(*)	sum(count(*)) over
'RSS'	'2020-06-22'	2	2
'RSS'	'2020-06-23'	3	5
'RSS'	'2020-06-24'	1	6
'Youtube'	'2020-06-22'	9	
'Youtube'	'2020-06-23'	1	
'Youtube'	'2020-06-24'	16	

```
select
  podcast.type,
  date_trunc('day', item.pub_date),
  sum(count(*)) over (
    partition by podcast.type
    order by date_trunc('day', item.pub_date)
    rows between unbounded preceding and current row
  )
from ...
```

podcast.type	date_trunc('day', item.pub_date)	count(*)	sum(count(*)) over
'RSS'	'2020-06-22'	2	2
'RSS'	'2020-06-23'	3	5
'RSS'	'2020-06-24'	1	6
'Youtube'	'2020-06-22'	9	9
'Youtube'	'2020-06-23'	1	10
'Youtube'	'2020-06-24'	16	26

And if I want to use  
**proprietary syntax**  
of my **favorite RDBMS** ?







# PostgreSQL

manage JSON with ease

with **JSONB**

```
CREATE TABLE public.podcast (  
  id uuid NOT NULL,  
  description character varying(65535),  
  has_to_be_deleted boolean,  
  last_update timestamp with time zone,  
  signature character varying(255),  
  title character varying(255),  
  type character varying(255),  
  url character varying(65535),  
  metadata jsonb DEFAULT '{}',  
  cover_id uuid  
);
```

Specific type of PostgreSQL



And a set of specific operators: ->, ->>, #>, #>>, @>, <@, and many more

## JSON stored as Binary data, indexable



96365a7c-8875-44f1-9f25-9b6fadbb4ec2	Goood Morning Web	<code>{}</code>
e4335987-9d2b-4e24-95fb-93164c9d3809	IFTTD - If This Then Dev	<code>{}</code>
cf615560-5f61-42f2-a931-0bf43c6823cd	JetBrainsTV	<code>{"youtube": {"channelId": "UC_JOQII0QJEF0QI"}}</code>
44d82b60-41b0-4e21-a252-c680ef366768	Message à caractère informatique	<code>{"youtube": {"channelId": "UC_OJQ0NOIHAAAIHIAJ"}}</code>
2e8a6a65-d1ab-4128-b355-edd5e11bddee	JUG Leaders	<code>{"youtube": {"channelId": "UC_QE5QFEQ6E45GH3AC5"}}</code>

And

JOOQ

can use it !

```

public Optional<YoutubePodcast> findOne(UUID id) {
    var youtubeField = jsonPath(PODCAST.METADATA, "youtube", "channelId");
    return query
        .select(PODCAST.ID, PODCAST.TITLE, PODCAST.URL, youtubeField)
        .from(PODCAST)
        .where(PODCAST.ID.eq(id))
        .and(hasKey(PODCAST.METADATA, "youtube"))
        .orderBy(PODCAST.ID.asc())
        .fetchOptional(it -> new YoutubePodcast(
            it.get(PODCAST.ID), it.get(PODCAST.TITLE),
            it.get(PODCAST.URL), it.get(youtubeField)
        ));
}

```

```

private static Field<String> jsonPath(Field<JSONB> jsonField, String... path) {
    return DSL.field("{0} #>> {1}", String.class, jsonField, DSL.array(path));
}

public static Condition hasKey(Field<JSONB> f, String... keys) {
    return DSL.condition("{0} ??| {1}", f, DSL.array(keys));
}

```

Custom JSONB operator from java code



```
fun findOne(id: UUID): YoutubePodcast? {  
    val youtubeField = PODCAST.METADATA.path("youtube", "channelId")  
    return query  
        .select(PODCAST.ID, PODCAST.TITLE, PODCAST.URL, youtubeField)  
        .from(PODCAST)  
        .where(PODCAST.ID.eq(id))  
        .and(PODCAST.METADATA.hasKey("youtube"))  
        .orderBy(PODCAST.ID.asc())  
        .fetchOne { (id, title, url, channelId) ->  
            YoutubePodcast(id, title, url, channelId)  
        }  
}
```

```
private fun Field<JSONB?>.path(vararg path: String): Field<String> {  
    return DSL.field("{0} #>> {1}", String::class.java, this, DSL.array(*path))  
}  
  
private fun Field<JSONB?>.hasKey(vararg keys: String): Condition {  
    return DSL.condition("{0} ??| {1}", this, DSL.array(*keys));  
}
```

Custom JSONB extension function from **Kotlin** code

```
select
  podcast.id, podcast.title, podcast.url,
  podcast.metadata #>> array['youtube', 'channelId']
from podcast
where (
  podcast.id = '2e8a6a65-d1ab-4128-b355-edd5e11bddee'
  and (podcast.metadata ??| array['youtube']))
)
```



You can leverage

SQL and NoSQL

from

PostgreSQL



Goodbye **MongoDB** !

This **talk** is  
about

JOOQ

But JOOQ is  
about

SQL

JOOQ  
is the best way to  
**learn SQL**

But why

SQL is so

**important ?**

SQL

is just

**Declarative**

SQL is

**everywhere !**





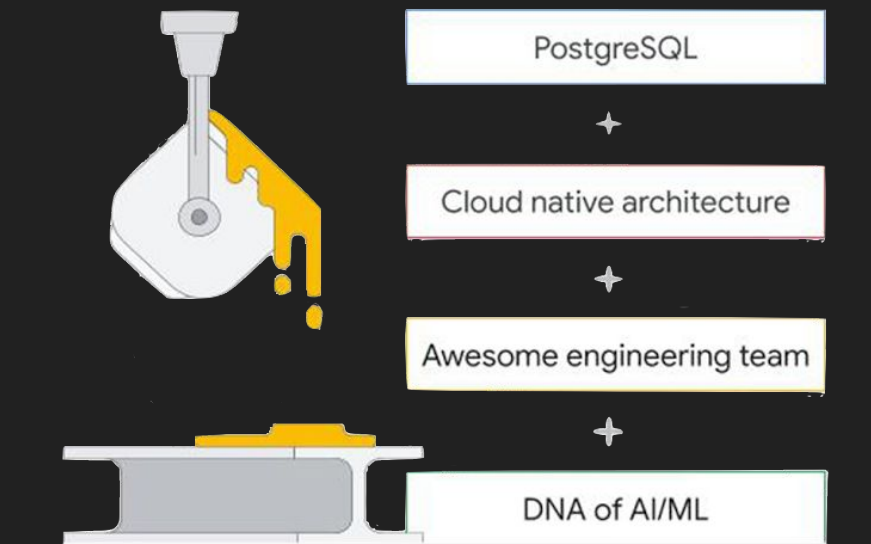
# Spanner

With **PostgreSQL** interface



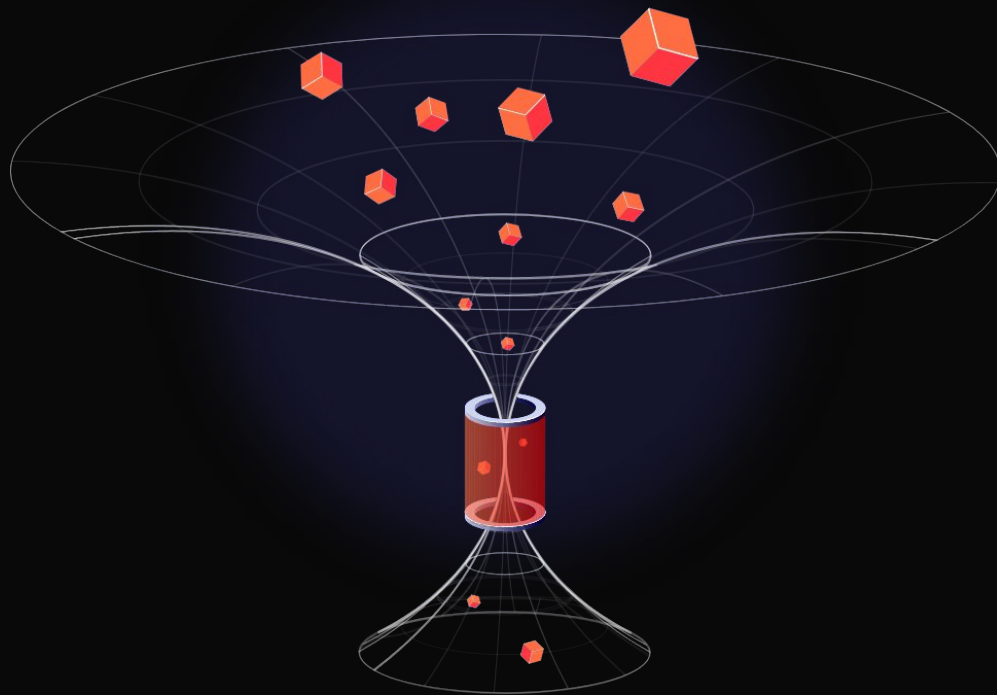


# AlloyDB for PostgreSQL





# yugabyte**DB**





BigQuery





**RedisSQL**



APACHE

DRILL



PrestoDB

# SQL for MongoDB



JET  
BRAINS





# SQL

is becoming


# the data language

JOOQ  
is **not** compatible  
with all of these (yet\*)

But using JOOQ

will help you to

**learn and understand SQL**

This is why  
I  JOOQ & SQL

DEVOXX France

5ème édition - 20 au 22 avril 2016 - Paris

Get Back in Control of your SQL

SQL and Java could work together so much better if we only let them.

#DevoxxFR Copyright (c) 2009-2016 by Data Geekery GmbH. Slides licensed under CC BY SA 3.0 DEVOXX France

[JOOQ, Get back in control of your SQL](#)

DEVOXX France

5ème édition - 20 au 22 avril 2016 - Paris

10 SQL tricks to convince you SQL is awesome

#DevoxxFR Copyright (c) 2009-2016 by Data Geekery GmbH. Slides licensed under CC BY SA 3.0 DEVOXX France

[10 SQL Tricks That You Didn't Think Were Possible](#)



More Than a Query Language: SQL in the 21st Century  
by **Markus Winand**



Thank you



Questions ?

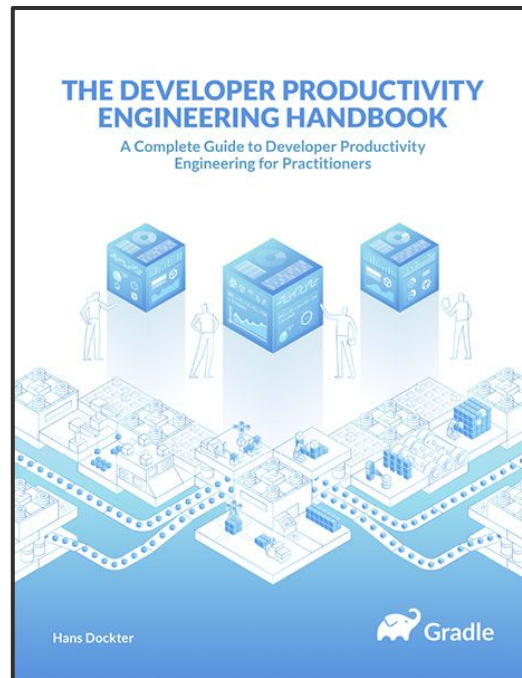




Stickers & Swag !



## GRADLE SWAG GIVEAWAY





KEEP  
SQL CALM  
AND  
USE  
SQL

```
var stmt = connection.prepareStatement("""
    SELECT ID, TITLE, URL
    FROM PODCAST
    ORDER BY ID ASC
""");
var rs = stmt.executeQuery();

var podcasts = new ArrayList<Podcast>();

while (rs.next()) {
    var id = rs.getObject("ID", UUID.class);
    var title = rs.getString("TITLE");
    var url = rs.getString("URL");

    podcasts.add(new Podcast(id, title, url));
}

rs.close();
stmt.close();

return podcasts;
```