ORACLE

Unleash the power of Java, JSON, and databases

CodeMash 2024

Josh Spiegel
Software Engineer
Oracle Database

January 11, 2024

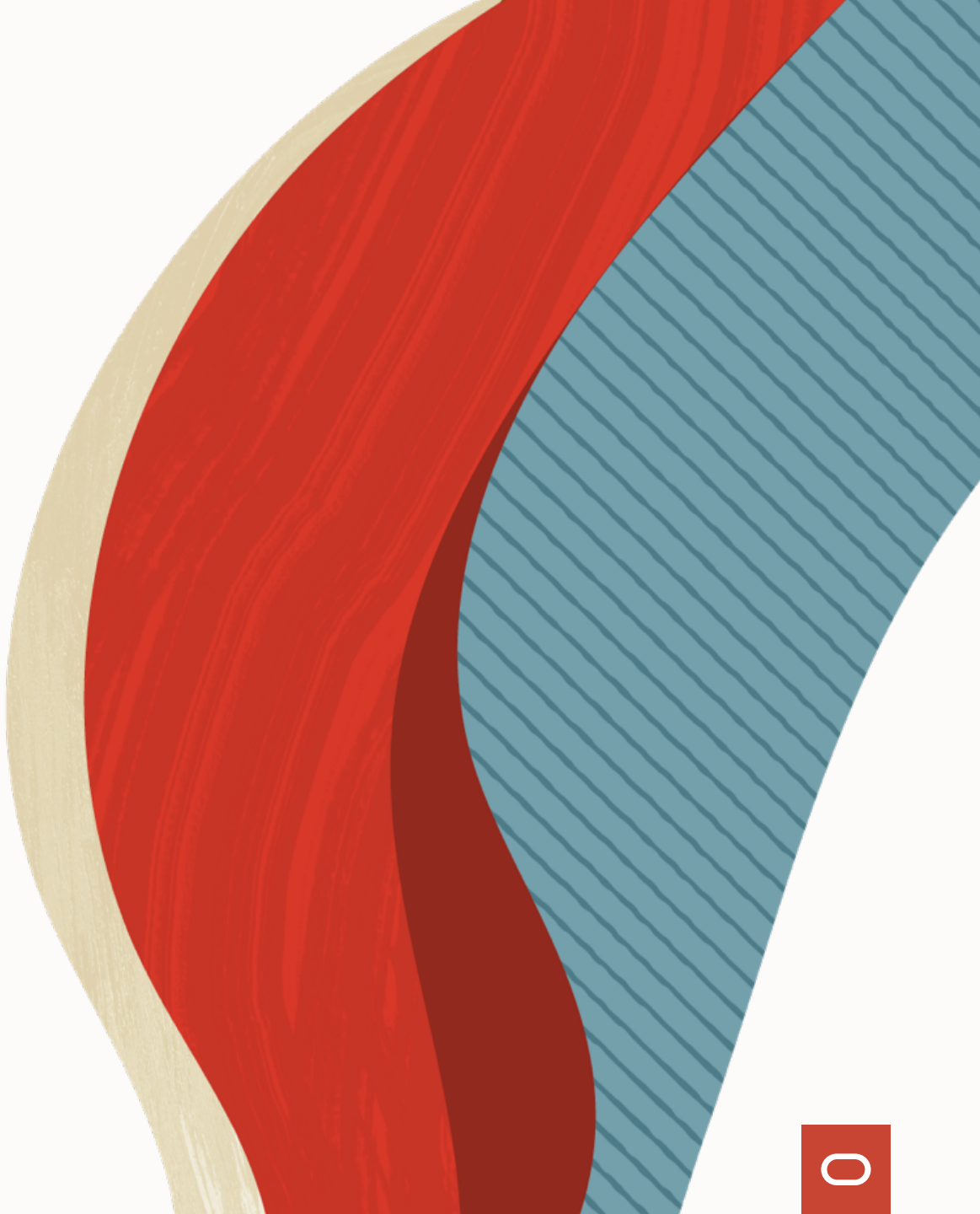
Outline

Introduction
Why store data as JSON?

SQL/JSON
Store and query JSON

JDBC/JSON
Access JSON in Java with SQL

Spring Data with JSON
Using Spring Data JDBC with Oracle JSON



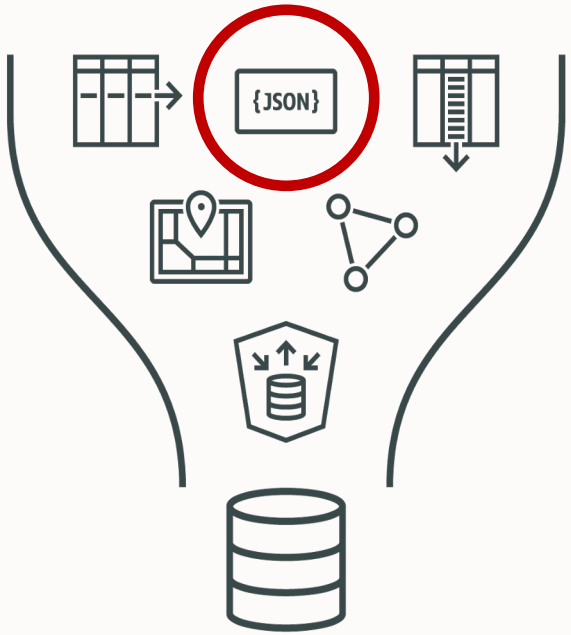
JSON Document Database

- Strings, numbers, Booleans arrays, and objects
- Objects/arrays can nest
- Schema independent

```
{
  "movie_id" : 1652,
  "title" : "Iron Man 2",
  "date" : "2010-05-07",
  "cast" : [
    "Robert Downey Jr.",
    "Larry Ellison"
  ]
}
```

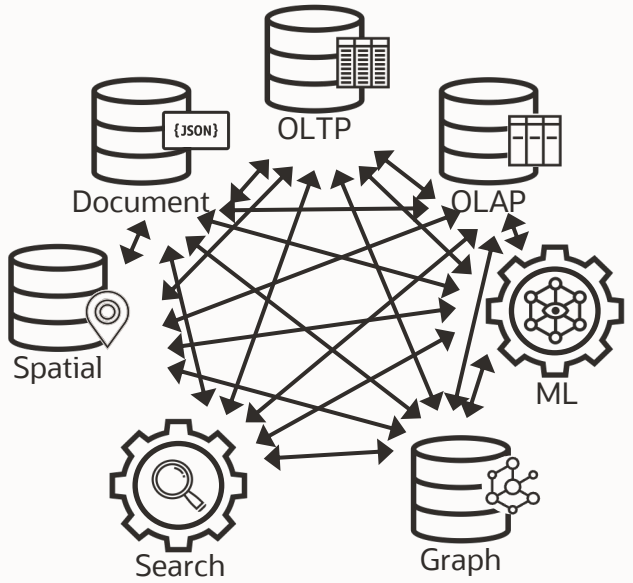
Oracle Converged Database

Converged Database Architecture



for **any** data type or workload

Single-purpose databases



for each data type and workload

Multiple security models, languages, skills, licenses, etc

Specialized versus MultiModel

Specialized Database Trends

IMS | Relational DBs | Object DBs | Document, Key-value, Columnar, etc. | Graph DBs | 'Next DB'

1960s | 1970s | 1980s | 1990s | 2000s | 2010s | 2020s | Future

Object Types

XML Type
SQL/XML
XQuery

JSON Type
SQL/JSON

SQL with graph

JSON Relational Duality

ORACLE

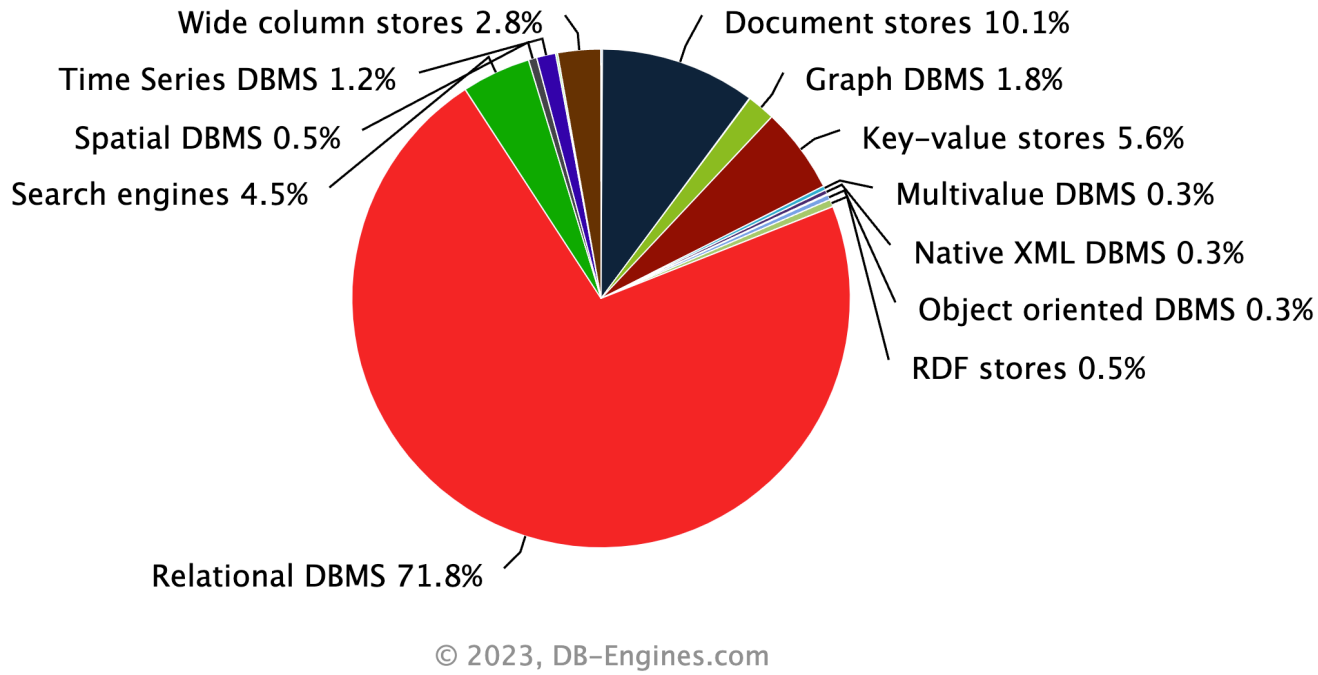
Multi-Model SQL

Database Popularity (db-engines.com)

Top-10 Ranked Databases

Database	Type	Popularity
Oracle	Relational, Multi-model	1231.48
MySQL	Relational, Multi-model	1163.94
Microsoft SQL Server	Relational, Multi-model	930.06
PostgreSQL	Relational, Multi-model	612.82
MongoDB	**Document**	**425.36**
Redis	**Key-value, Multi-model**	**167.35**
IBM Db2	Relational, Multi-model	144.89
Elasticsearch	**Search engine, Multi-model**	**143.75**
Microsoft Access	Relational	134.45
SQLite	Relational	131.21

Total Popularity % by Type



Why store data as JSON documents?

```
{
  "movie_id" : 1652,
  "title" : "Iron Man 2",
  "date" : "2010-05-07",
  "cast" : [
    "Robert Downey Jr.",
    "Larry Ellison",
    ...
  ]
}
```

```
class Movie {

  int movie_id;
  String title;
  LocalTime date;
  List<String> cast;

  Movie() {
   ...
}
```

```
db.movies.insertOne(
  movieValue
);

db.movies.find(
   {"movie_id" : 1653 }
);
```

Schema-flexible

- No upfront schema design
- Application-controlled schema
- Simple data model

Less Impedance Mismatch

- Maps to application objects
- Supports nested structures
- Read/write without joins

"No SQL"

- No embedding of SQL code
- Minimal new concepts for developers

NoSQL Document Store Features

Elastic compute and storage

Single-digit latency reads and writes

Highly available

Low-cost

Our goal: bridging the Gap between **JSON Document Store** and **Relational Worlds**

{JSON}

Schema-flexible data
Low-latency, High throughput
Scalable, Highly available
Low-cost, developer friendly
MongoDB compatible

SQL

SQL and analytics
In-memory columnar
Resource management
Data Normalization
ACID

SQL/JSON

Oracle Database JSON

Schema-flexible JSON stored within a structured column

Defined by **ISO SQL 2016, 2023**

Stored using query-efficient native binary JSON format OSON

```
CREATE TABLE movies (data JSON);

INSERT INTO movies VALUES (
 JSON {
   'id'      : 123,
   'title'   : 'Iron Man'
 }
);

SELECT m.data.title
FROM movies m;
```

SQL/JSON

- Use SQL to query JSON data
  - JSON to relational
  - Relational to JSON
- Joins, aggregation, projection
- Construct new JSON values
- Update JSON values
- Unnest arrays

JSON aggregation and construction

```
SELECT JSON {
 'total' : sum(t.data.gross),
 'genre' : m.data.genre
}
FROM movies m
GROUP BY e.data.genre
```

JSON unnesting

```
SELECT title, actor
FROM movies NESTED data COLUMNS (
  title,
  NESTED actors[*] COLUMNS (
    actor
  )
)
WHERE id = 123
```

JSON Storage History

12c (2013)

JSON-text storage
and query processing
(clob, blob, varchar2)

18c

19c

Binary JSON storage (OSON) and
Mongo API support added for
Autonomous Databases
(in BLOB columns)

21c

Native JSON datatype and
collections backed by OSON
(all database types)

Why OSON? - Extended Types

Standard

- OBJECT { }
- ARRAY [ ]
- STRING
- TRUE/FALSE
- NULL
- NUMBER

Extended

- BINARY_FLOAT
- BINARY_DOUBLE
- TIMESTAMP/DATE
- INTERVALDS/INTERVALYM
- RAW

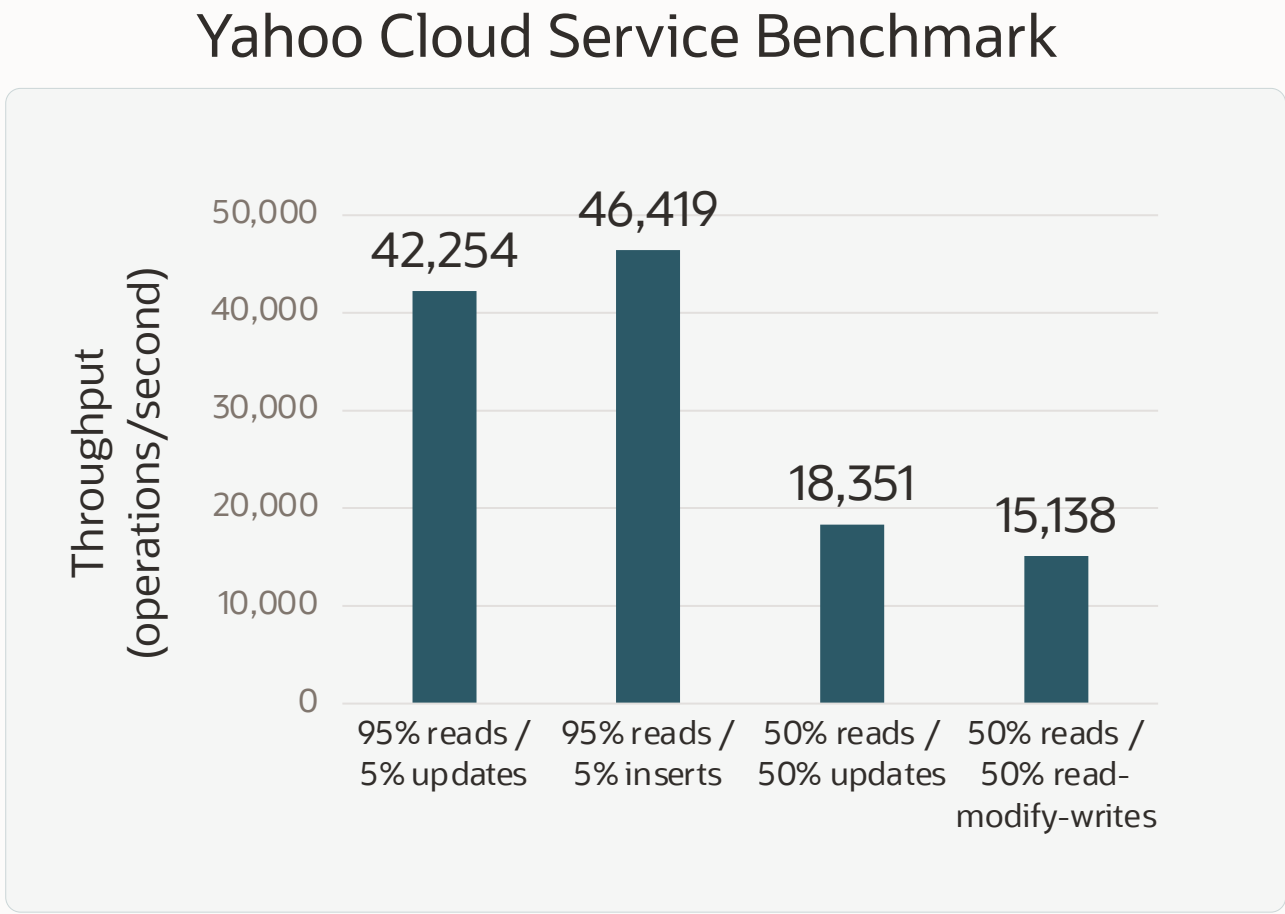
Fidelity with relational data

```
CREATE TABLE orders VALUES (
    oid         NUMBER,
    created     TIMESTAMP,
    status      VARCHAR2(10),
);


{
  "oid":123,
  "created":"2020-06-04T12:24:29Z",
  "status":"OPEN"
}
```

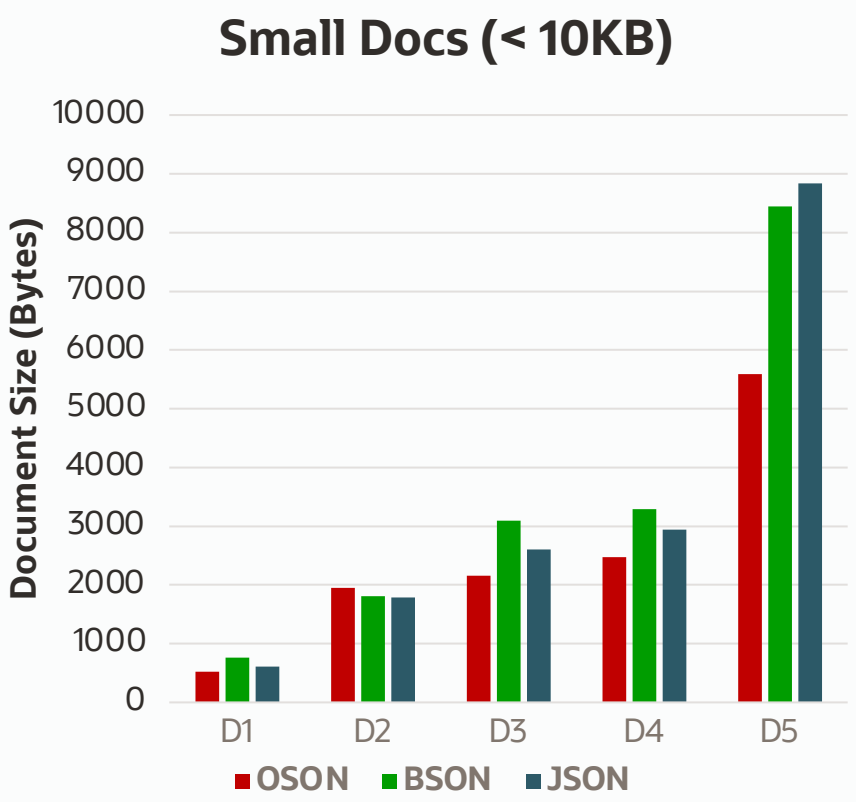
Why OSON? - Performance

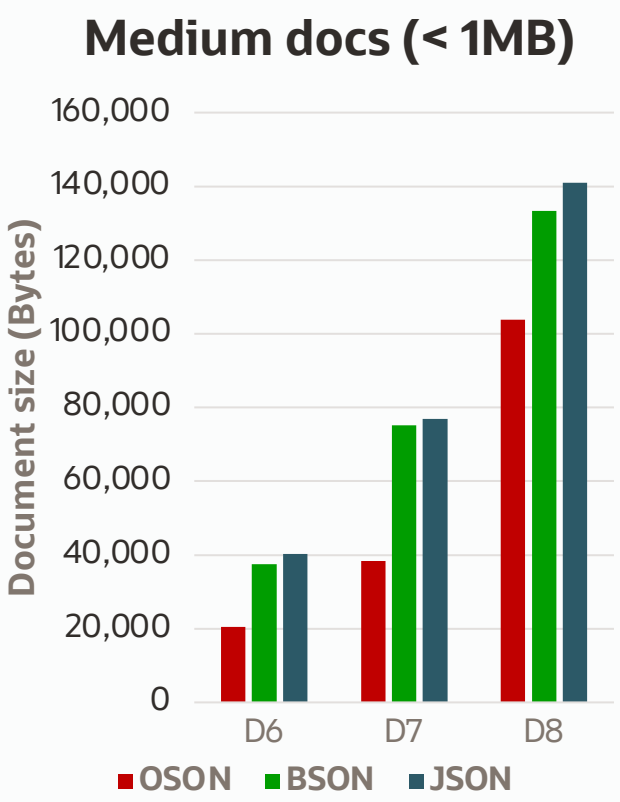
- Faster path evaluation
- Efficient random access
- Smaller than JSON text and BSON
- Less network and storage IO
- No text parsing or serialization

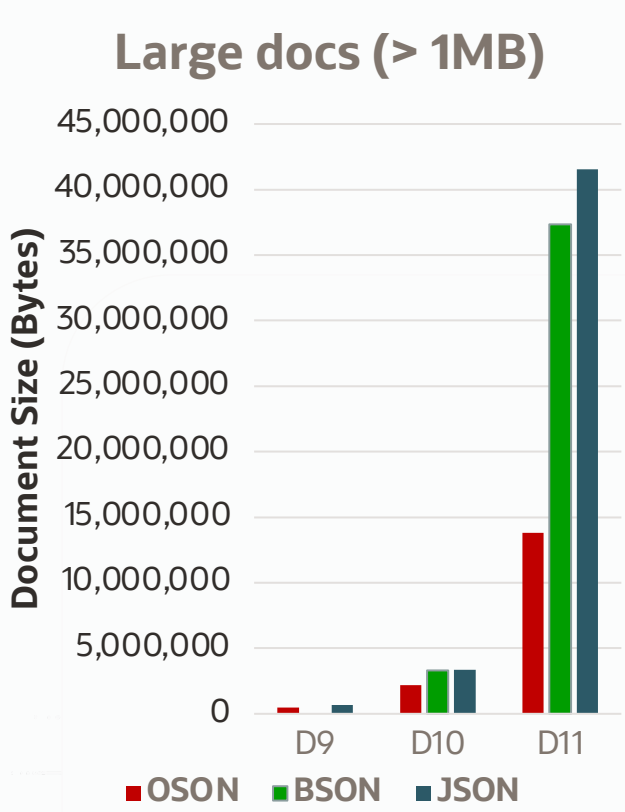


See VLDB 2020:
Native JSON Datatype Support: Maturing SQL and NoSQL convergence in Oracle Database

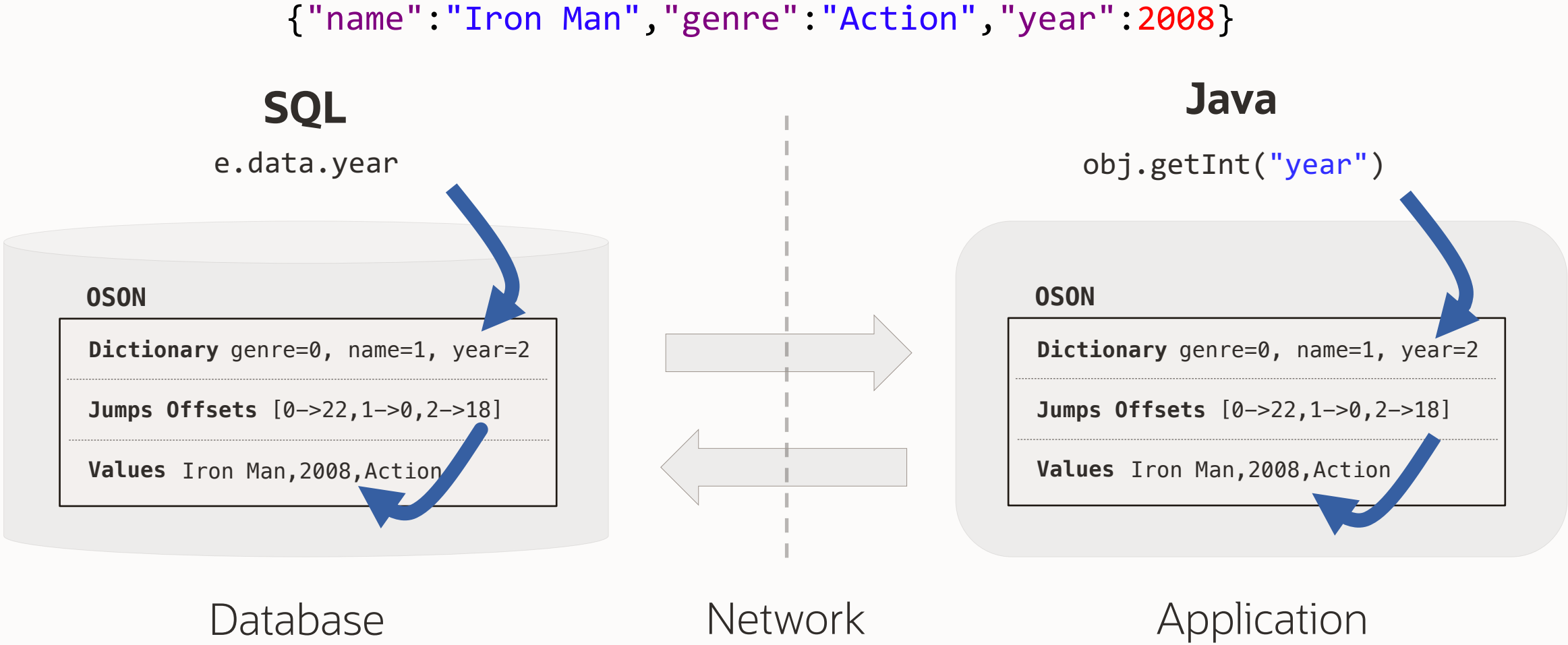
Why OSON? - Performance







OSON Performance



JDBC/JSON

Modeling JSON in Java (`oracle.sql.json`)

Binding and reading JSON from a statement

The Java API for JSON in JDBC (21c+)

Facilities to read, write, and modify binary JSON values from Java.

Features

- Mutable tree/object model
- Event-based parser and generator
- Access to extended SQL/JSON types (TIMESTAMP, DATE, etc.)
- Supports both JSON text and OSON
- Optional integration with JSON-P / JSR-374

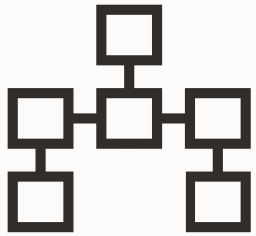
Where is it?

JAR: ojdbc11.jar

- OTN
- Maven Central Repository
  (groupId = `com.oracle.database.jdbc`, artifactId = `ojdbc11`)

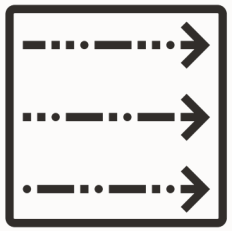
Package: `oracle.sql.json`

Package `oracle.sql.json`



Tree Model

`OracleJsonObject`
`OracleJsonArray`
`OracleJsonString`
`OracleJsonDecimal`
`OracleJsonDouble`
`OracleJsonTimestamp`
`OracleJsonBinary`
`...`



Event Model

`OracleJsonParser`
`OracleJsonGenerator`

Factory

`OracleJsonFactory`

OracleJsonFactory Methods

Read/write **OSON**

```
createJsonBinaryGenerator(OutputStream)
createJsonBinaryValue(ByteBuffer)
createJsonBinaryValue(InputStream)
createJsonBinaryParser(ByteBuffer)
createJsonBinaryParser(InputStream)
```

Read/write **JSON text**

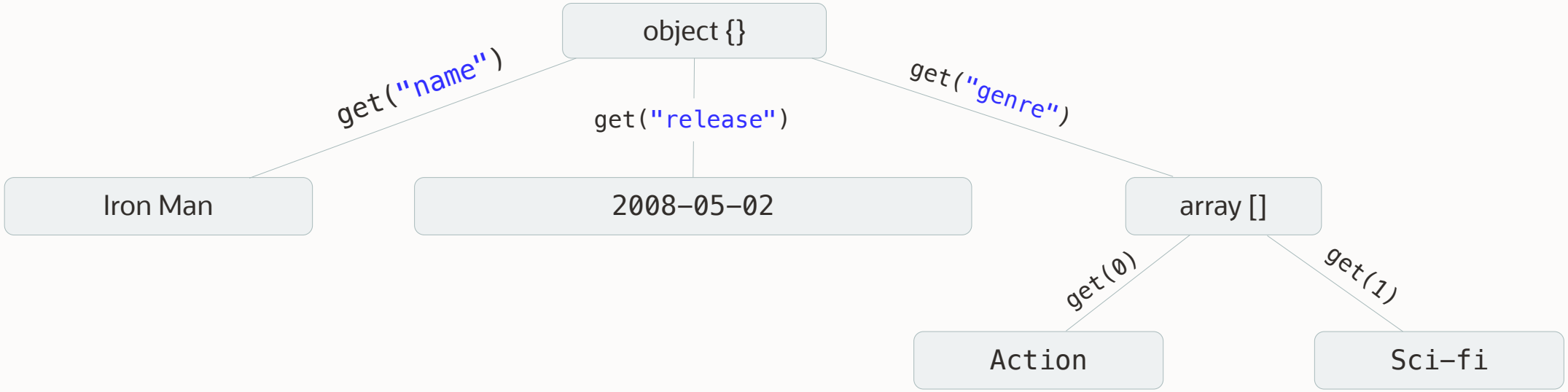
```
createJsonTextGenerator(OutputStream)
createJsonTextGenerator(Writer)
createJsonTextValue(InputStream)
createJsonTextValue(Reader)
createJsonTextParser(InputStream)
createJsonTextParser(Reader)
```

In-memory **tree model** creation

```
createObject()
createObject(OracleJsonObject)
createArray()
createArray(OracleJsonArray)
createString(String)
createDecimal(BigDecimal)
createDecimal(int)
createDecimal(long)
createDouble(double)
createTimestamp(Instant)
....
```

Tree Model

```
{
  "name" : "Iron Man",
  "release" : "2008-05-02",
  "genre" : ["Action","Sci-fi"]
}
```



Creating a value

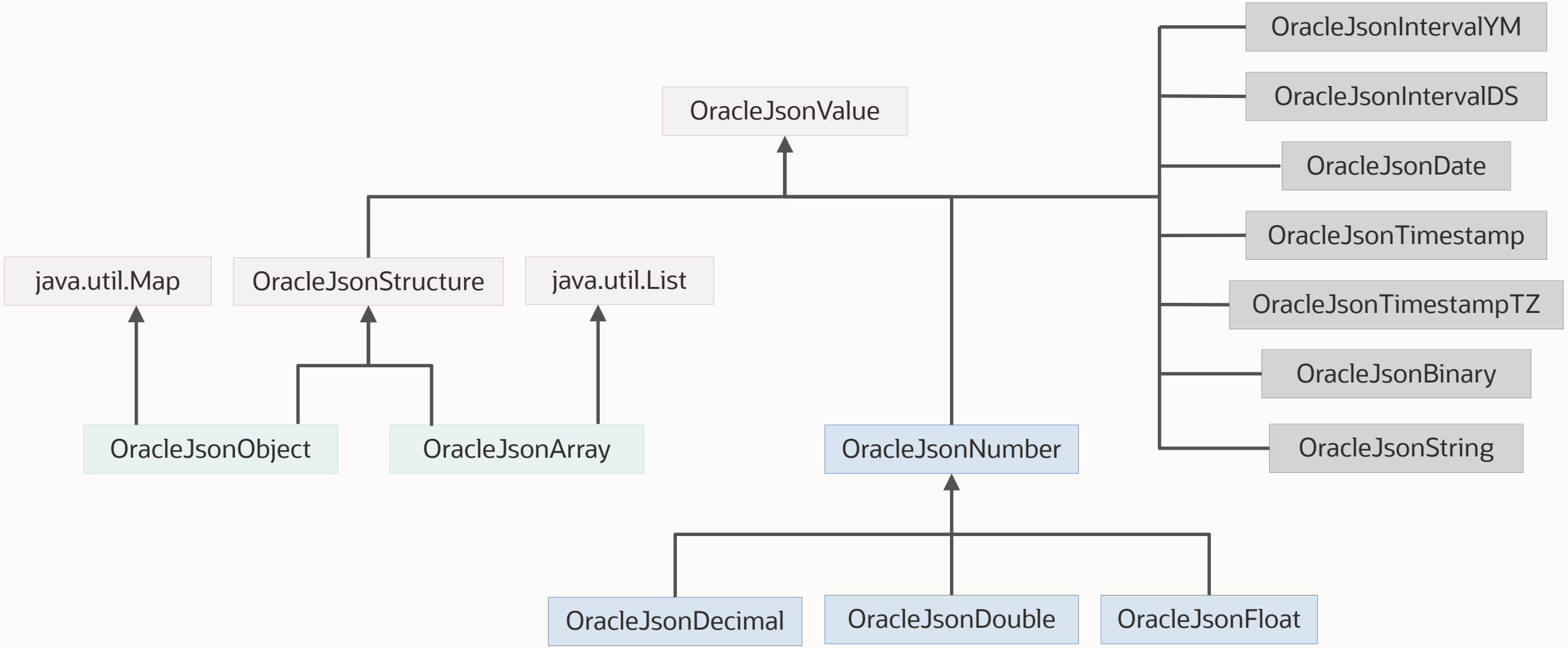
```
OracleJsonFactory factory = new OracleJsonFactory();

OracleJsonObject movie = factory.createObject();
movie.put("name", "Iron Man");
movie.put("year", 2008);
movie.put("created", Instant.now());

System.out.println(movie.get("name"));
System.out.println(movie.toString());
```

```
Iron Man
{"name":"Iron Man","year":2008,"created":"2023-09-21T21:25:05.610718"}
```

Tree Model – Type Hierarchy



Event Model

```
{
  "name" : "Iron Man",
  "created" : "2023-09-21T21:25:05.610718",
  "genre" : ["Action", "Adventure"]
}
```

`OracleJsonParser`

- Produces a sequence of events
- `parser.next()`
- Events can come from text or OSON

`OracleJsonGenerator`

- Consumes a sequence of events
- `generator.write(...)`
- Generates text or OSON

SQL Integration

- **SQL** statements can **consume** and and **produce** JSON values
- Relies on standard `getObject()` and `setObject()` methods throughout JDBC

Getting JSON from the database

SQL Methods that return JSON

`java.sql.ResultSet`
`  .getObject(*, Class<T>)`
 	
`java.sql.CallableStatement`
`  .getObject(*, Class<T>)`

Class	Content
`java.lang.String, java.io.Reader` `java.io.InputStream`	`JSON text`
`oracle.sql.json.OracleJsonValue` `jakarta.json.JsonValue`	`Tree Model`
`oracle.sql.json.OracleJsonParser` `jakarta.json.stream.JsonParser`	`Event Model`
`oracle.sql.json.OracleJsonDatum`	`Raw OSON`

```
ResultSet rs = stmt.executeQuery("SELECT data FROM movies");
rs.next();
OracleJsonObject movie = rs.getObject(1, OracleJsonObject.class);
```

Sending JSON to the database

SQL methods accept JSON
java.sql.PreparedStatement **.setObject**(*, Object, SQLType) java.sql.ResultSet **.updateObject**(*, Object, SQLType) java.sql.CallableStatement **.setObject**(*, Object, SQLType) java.sql.RowSet **.setObject**(*, Object, SQLType)

Class	Content
java.lang.CharSequence, java.lang.String java.io.Reader	JSON text
java.io.InputStream, byte[]	UTF8 or OSON
oracle.sql.json.OracleJsonValue jakarta.json.JsonValue	Tree Model
oracle.sql.json.OracleJsonParser jakarta.json.stream.JsonParser	Event Model
oracle.sql.json.OracleJsonDatum	Raw OSON

```
OracleJsonObject movie = ...;
PreparedStatement pstmt = con.prepareStatement("INSERT INTO movies VALUES (:1)");
pstmt.setObject(1, movie, OracleType.JSON);
```

JDBC/JSON Performance Tips

- Reuse **`OracleJsonFactory`** instances across requests. Thread local instances are best, a global instance is OK as it uses lockless thread safe data structures.
- Don't convert to and from **JSON text** if you don't have to!
  - Access values as `OracleJsonValue`, `OracleJsonParser, JSONP, JSONB` rather than `String`, `Reader`, `InputStream`, etc.
- When binding JSON (e.g. `setObject()`) specify **`OracleTypes.JSON`** as the third argument. If the driver does not know that the binding is JSON, it will let the server do the OSON encoding in some cases.
- Use **universal connection pooling** (UCP)
  - Creating new connections is expensive
  - Threads can share a connection at the same time, but it is not efficient to do so

  https://docs.oracle.com/en/database/oracle/oracle-database/20/jjucp/optimizing-ucp-behavior.html#GUID-FFCAB66D-45B3-4D7B-991B-40F1480630FD
- Turn on **statement caching** and use bind variables

Demo

Use SQL/JSON from Java using `oracle.sql.json`

Demo Source Available:

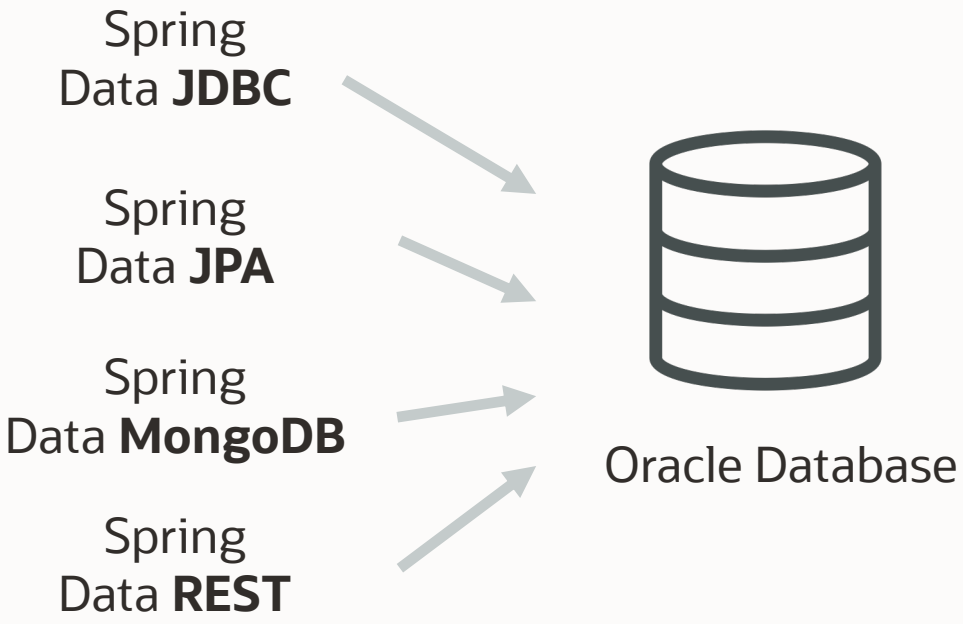
https://github.com/oracle/json-in-db/tree/master/JdbcExamples

Spring Data and JSON

Using Spring Data with JSON storage in Oracle Database

Spring Data

Provides a consistent programming model across different data stores.



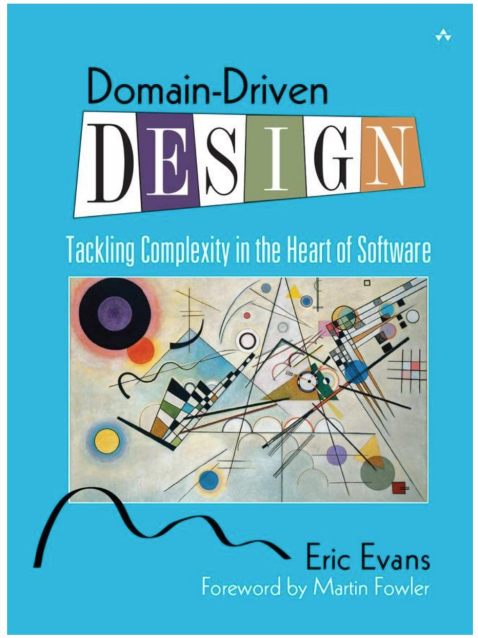

```java
public class Movie {
  @Id
  Integer id;
  String name;
  String genre;
}
```

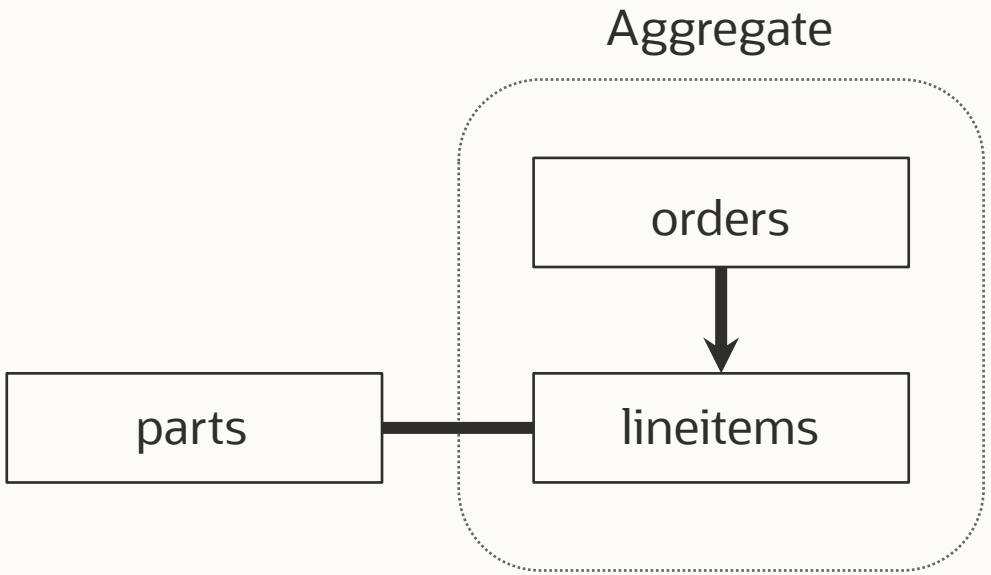
```java
Movie m = movies.findById(1).get();
System.out.println(
   m.getName() + " " + m.getId()
);
```

Oracle Database is compatible with multiple different spring data adapters.

*Interact with your business objects as a repository of objects (**no SQL or JSON directly**)*

Spring Data JDBC





- Spring Data JDBC implemented directly over JDBC for Oracle, MySQL, Postgres, etc.
- Inspired by domain driven design
- Repository models a collection of **aggregates**
- Aggregate root and nested entities change as a single unit of change

Spring Data JDBC Example

```java
@Data
public class Movie {
  @Id
  Integer id;
  String name;
  List<Image> images;
}
```

```java
@Data
public class Image {
  String file;
  String description;
}
```


MOVIES

ID	NAME
1	Iron Man
2	Interstellar
3	The Matrix

IMAGE

FILE	DESCRIPTION	MOVIE	MOVIE_KEY
img1.png	Robert Downey Jr.	1	0
img2.png	Gwyneth Paltrow	1	1
img3.png	Keanu Reeves	3	0
img4.png	Matthew McConaughey	2	0

Spring Data JDBC Example - findById

```java
@Data
public class Movie {
  @Id
  Integer id;
  String name;
  List<Image> images;
}

@Data
public class Image {
  String file;
  String description;
}

public interface MovieRepository
   extends CrudRepository<integer, Movie>

}
```

```java
Movie ironMan = movieRepo.findById(1);
```

```sql
SELECT
   "MOVIE"."ID" AS "ID",
   "MOVIE"."NAME" AS "NAME"
FROM "MOVIE"
WHERE "MOVIE"."ID" = :1

SELECT
  "IMAGE"."FILE" AS "FILE",
  "IMAGE"."DESCRIPTION" AS "DESCRIPTION",
  "IMAGE"."MOVIE_KEY" AS "MOVIE_KEY"
FROM "IMAGE"
WHERE "IMAGE"."MOVIE" = :1
ORDER BY "MOVIE_KEY"
```

Default behavior, could be overridden with explicit join query.

Spring Data JDBC Example – save()

```java
@Data
public class Movie {
  @Id
  Integer id;
  String name;
  List<Image> images;
}

@Data
public class Image {
  String file;
  String description;
}

public interface MovieRepository
   extends CrudRepository<integer, Movie>

}
```

```java
movieRepo.save(newMovie);
```

```
INSERT INTO "MOVIE" ("NAME")
VALUES (:1)
RETURNING id INTO :2


INSERT INTO "IMAGE" (
  "DESCRIPTION", "FILE", "MOVIE", "MOVIE_KEY"
)
VALUES (:1, :2, :3, :4)
```

JSON is great fit for aggregates!

Single select, single insert
- Fewer roundtrips
- Less index maintenance
- Less random IO, block writes

ID	NAME	IMAGES
1	Iron Man	[{"file":"img1.png"}...]
2	Interstellar	[{"file":"img3.png"}...]
3	The Matrix	[{"file":"img4.png"}...]

Schema flexibility
- No secondary table creation
- Less "agreement" between Java and SQL required
- Attributes can change over time without modifying table definitions

Data stored in database "looks like" data in the application.

Less random IO, round-trips by default when modifying an aggregate

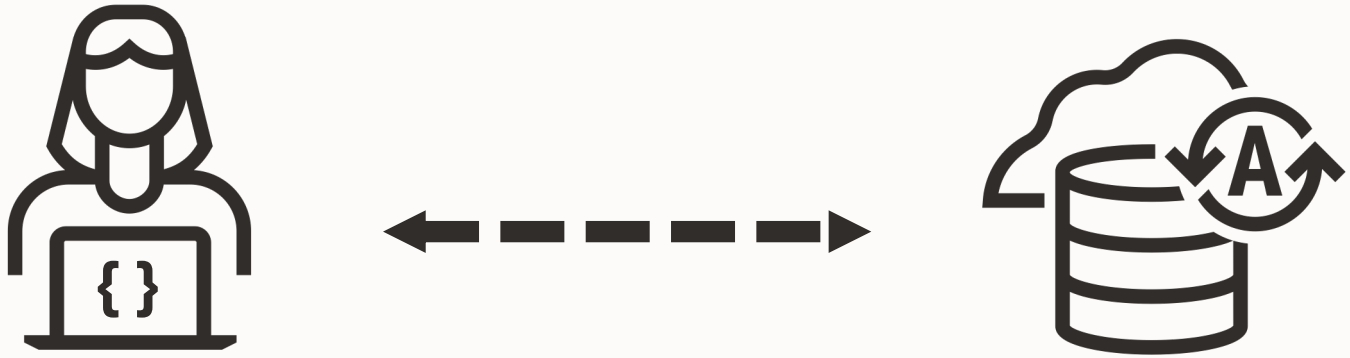
Demo

Using Spring Data JDBC with Oracle JSON

Demo Source Available:

https://github.com/oracle/json-in-db/tree/master/JdbcExamples/SpringDataJdbc

Oracle API for MongoDB



MongoDB Application **Oracle Database**

MongoDB developers keep using same skills, tools, and frameworks
Simplifies migrations from MongoDB to Oracle
MongoDB does not have tables – it stores collections of JSON documents

Thank you!

Join us on Slack

After joining **`oracledevrel`**, find us in the channel

`#oracle-db-json`

JSON Collections

NoSQL-style access to JSON in Oracle Database

JSON Collections

A document is a JSON value

Structure is flexible

A collection contains documents

Supports insert, get, update, filter

A database contains collections

Access data programmatically – "No SQL"

MongoDB Example

```
MongoClient mongoClient = MongoClients.create(connString);
MongoDatabase database = mongoClient.getDatabase("admin");

MongoCollection<Document> coll =
  database.createCollection("movies");

Document movie = Document.parse(json);
coll.insertOne(movie);

Bson filter = eq("title", "Iron Man");
MongoCursor<Document> cursor = coll.find(filter).cursor();
Document doc = cursor.next();
```

JSON Collections

Database => Schema

Collections created in database "admin" will be in the "ADMIN" schema

MongoDB Example

```
MongoClient mongoClient = MongoClients.create(connString);
MongoDatabase database = mongoClient.getDatabase("admin");

MongoCollection<Document> coll =
  database.createCollection("movies");

Document movie = Document.parse(json);
coll.insertOne(movie);

Bson filter = eq("title", "Iron Man");
MongoCursor<Document> cursor = coll.find(filter).cursor();
Document doc = cursor.next();
```

JSON Collections

Collection => Table

Collections are an abstraction or view of a table with a single JSON column.

```
create table movies
(
   ID VARCHAR2,
   DATA JSON
)
```

MongoDB Example

```
MongoClient mongoClient = MongoClients.create(connString);
MongoDatabase database = mongoClient.getDatabase("admin");

MongoCollection<Document> coll =
  database.createCollection("movies");

Document movie = Document.parse(json);
coll.insertOne(movie);

Bson filter = eq("title", "Iron Man");
MongoCursor<Document> cursor = coll.find(filter).cursor();
Document doc = cursor.next();
```

JSON Collections

Document => Row

Inserting a document into a collection inserts a row into the backing table.

```
insert into
  movies (data)
  values (:1);
```

MongoDB Example

```
MongoClient mongoClient = MongoClients.create(connString);
MongoDatabase database = mongoClient.getDatabase("admin");

MongoCollection<Document> coll =
  database.createCollection("movies");

Document movie = Document.parse(json);
coll.insertOne(movie);

Bson filter = eq("title", "Iron Man");
MongoCursor<Document> cursor = coll.find(filter).cursor();
Document doc = cursor.next();
```

JSON Collections

Filter => Query

Filter expressions are executed as SQL over the backing table. Fully utilizes core Oracle Database features such as indexing, cost-based optimization, etc.

```
select data
from movies e
where
  e.data.title = 'Iron Man'
```

MongoDB Example

```
MongoClient mongoClient = MongoClients.create(connString);
MongoDatabase database = mongoClient.getDatabase("admin");

MongoCollection<Document> coll =
  database.createCollection("movies");

Document movie = Document.parse(json);
coll.insertOne(movie);

Bson filter = eq("title", "Iron Man");
MongoCursor<Document> cursor = coll.find(filter).cursor();
Document doc = cursor.next();
```

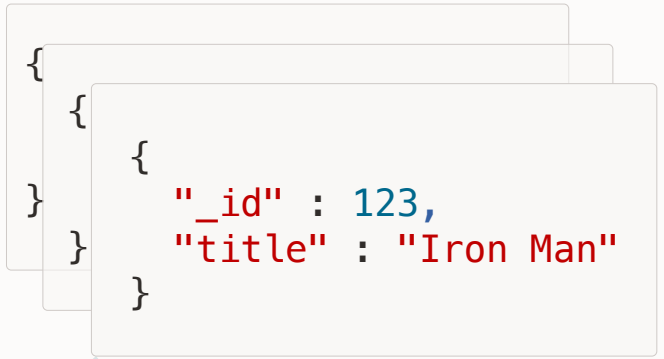
Oracle Database JSON

Natively stored JSON, SQL when you need it.

MongoDB Compatible APIs

```
movies.insertOne({
  "_id" : 123,
  "title"  : "Iron Man"
});
```

Schema-flexible storage, simple APIs



movies

SQL/JSON

```
select t.data.title
from movies t
where t.data._id = 123;
```

SQL analytics and reporting over natively stored JSON