ORACLE

DatabaseWorld at CloudWorld

Best Practices for Modern Application Development Using JSON and Java

LRN2977

Josh Spiegel
Software Architect
Oracle Database

Moumita Bardhan
Senior Manager
Renesas Electronics

Sriram Ganapathy
Team Lead
Renesas Electronics

Outline

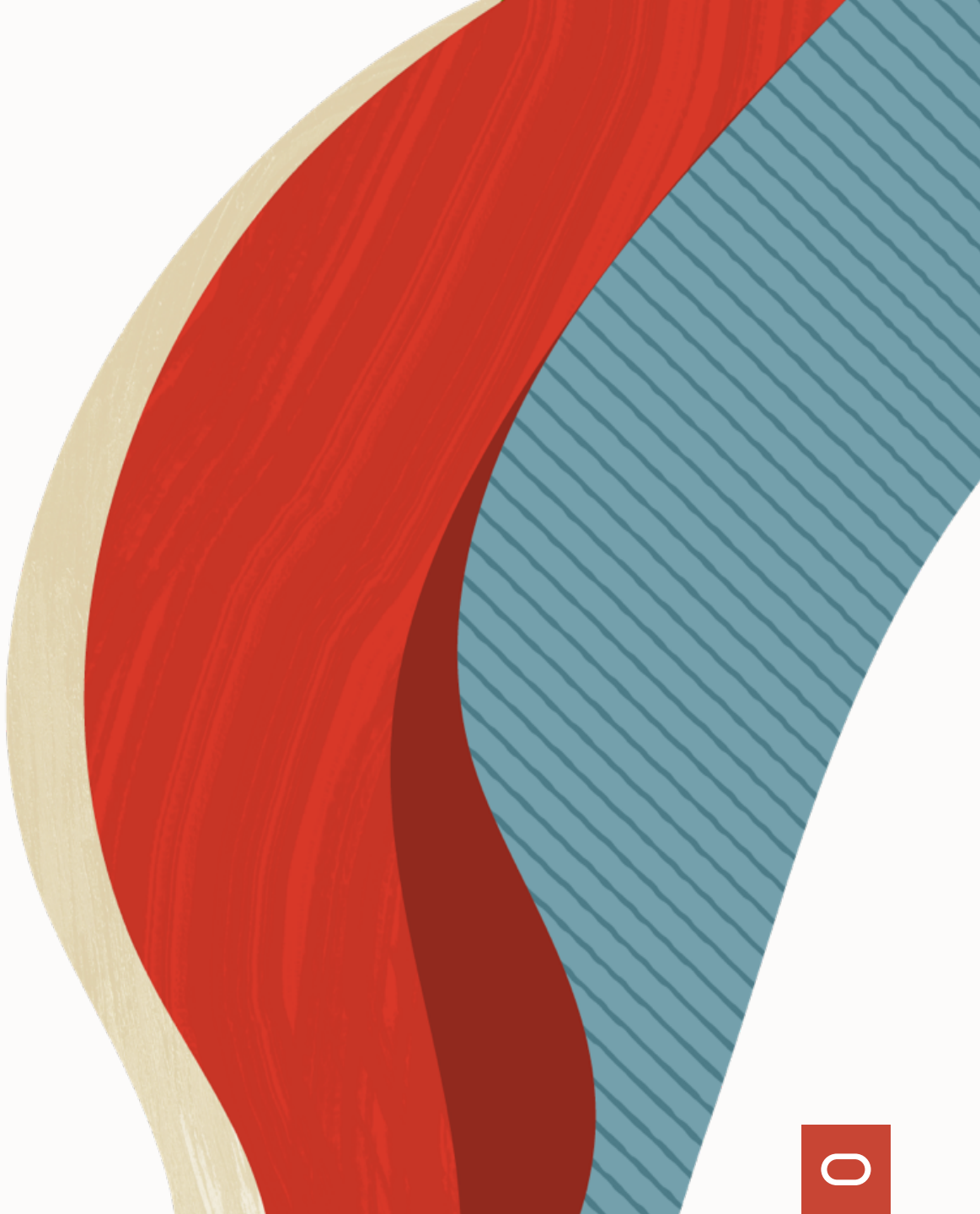
Introduction
Why store data as JSON?

SQL/JSON
Store and query JSON in Oracle Database

JDBC/JSON
Access JSON in Java with SQL

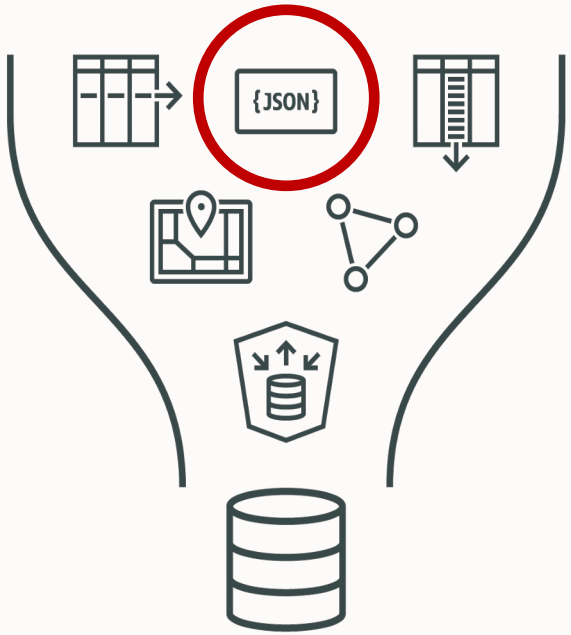
Spring Data with JSON
Using Spring Data JDBC with Oracle JSON

Renesas and JSON
Customer use case presentation



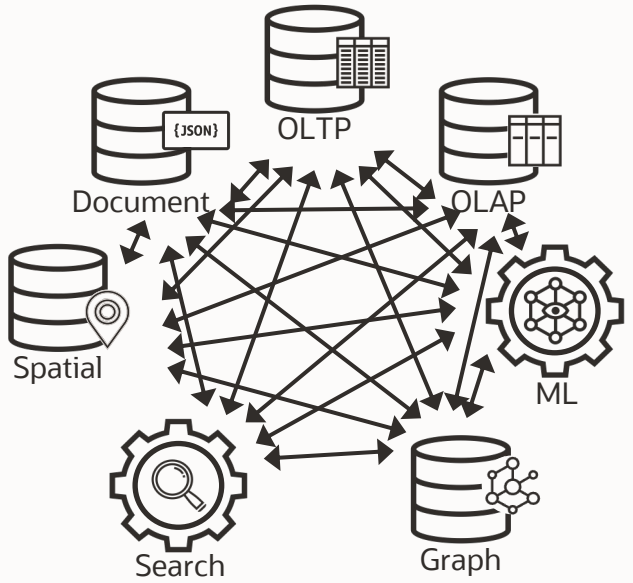
Oracle Converged Database

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

Storing Data as JSON

```
{
  "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

Oracle Database – has the features of a NoSQL JSON document store

- ✓ Low-latency
- ✓ High throughput
- ✓ Scalable compute

- ✓ Scalable storage
- ✓ Highly available
- ✓ Schema-flexible JSON

- ✓ SQL optional
- ✓ Collection APIs
- ✓ Schema-flexible JSON

With all the traditional benefits of Oracle Database on JSON documents…

- ✓ ANSI SQL / JSON
- ✓ Advanced analytics
- ✓ In-memory columnar
- ✓ Full-text search
- ✓ ACID transactions
- ✓ Consistency and durability
- ✓ Mission critical use cases
- ✓ Secure by default
- ✓ Resource management

SQL/JSON

JSON Columns

- Schema-flexible JSON stored in a structured column
- SQL extended to process JSON column values
- Stored using query-efficient OSON binary format
- ISO/SQL 2016, 2023

```
CREATE TABLE movies (data JSON);

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

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.jcol.gross.number()),
 'genre' : m.jcol.genre.string()
}
FROM movies m
GROUP BY e.jcol.genre.string()
```

JSON unnesting

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

JSON Storage History

12c

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

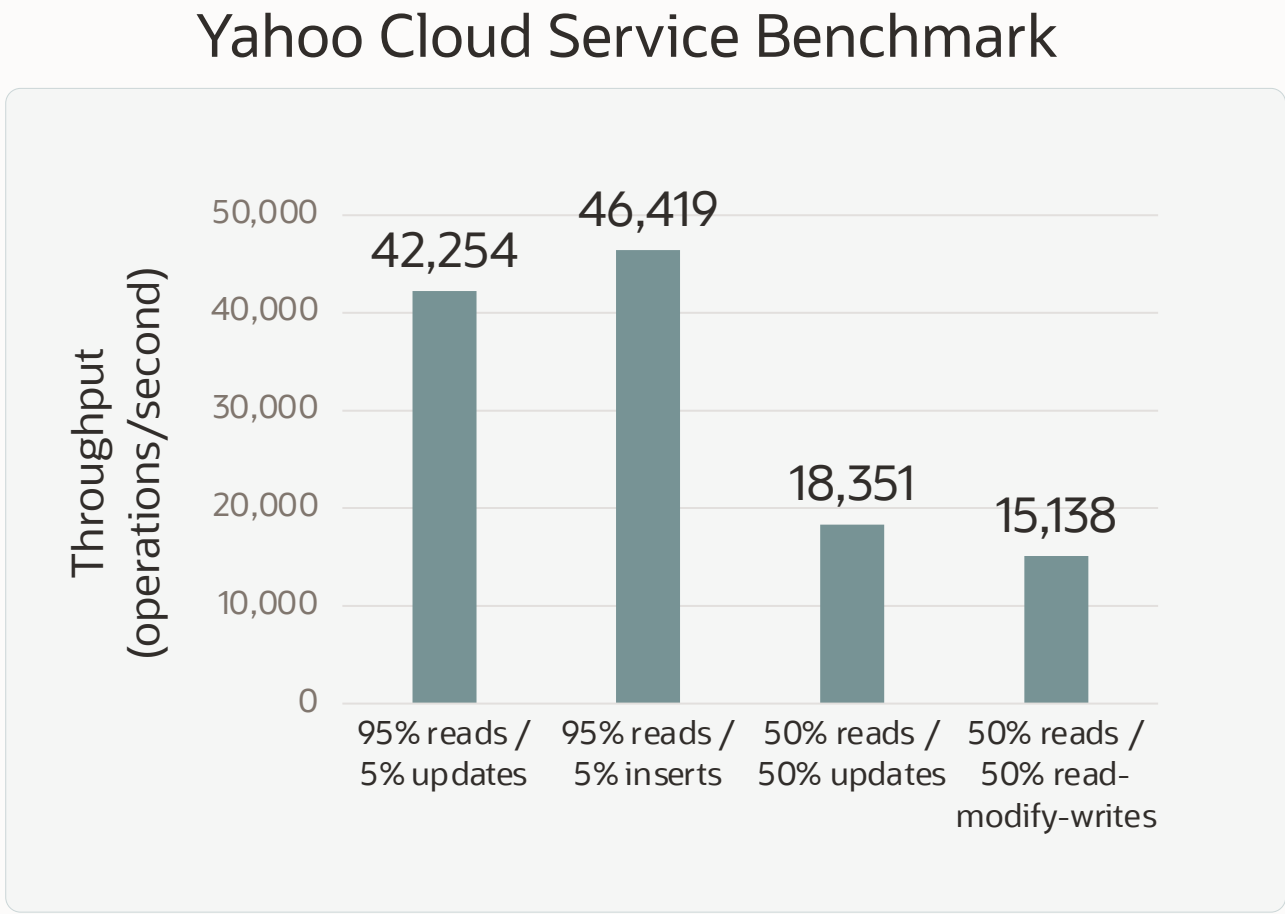
Fidelity with relational data

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


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

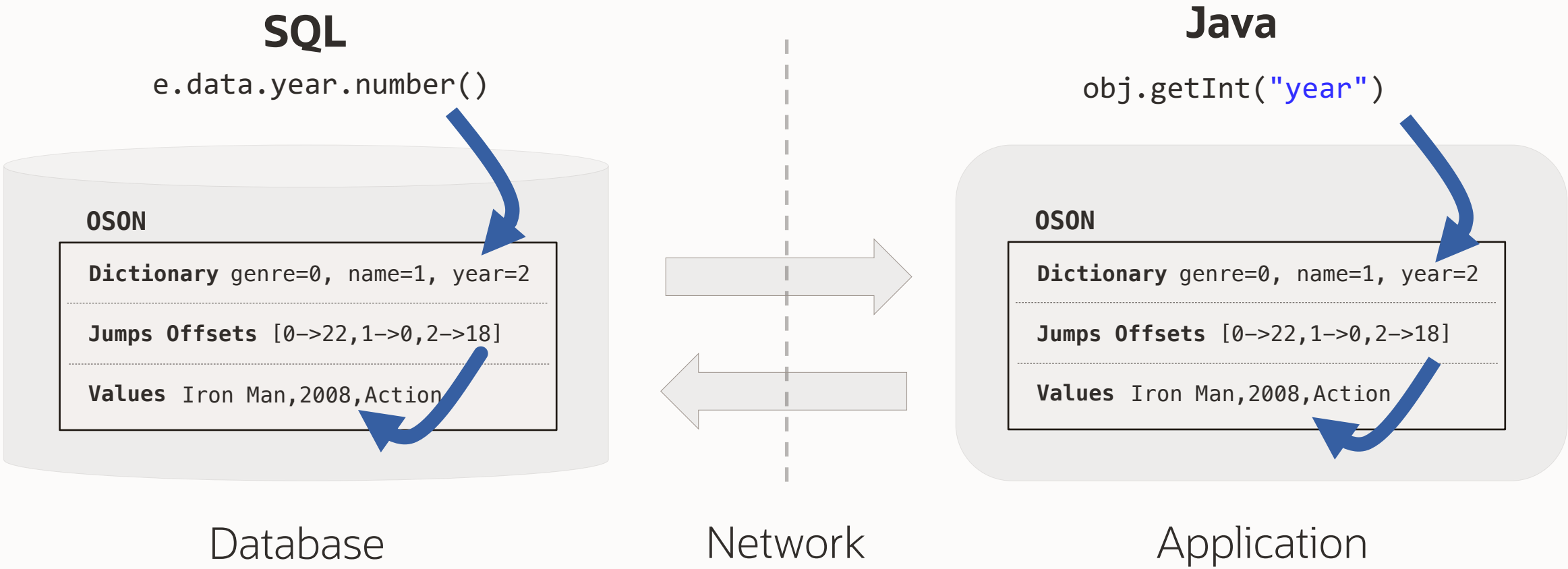
Why OSON? - Performance

- 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

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

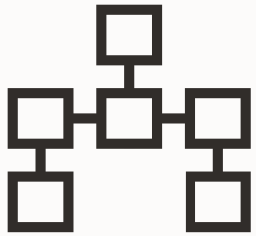
Where is it?

JAR: ojdbc11.jar

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

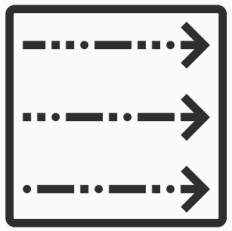
Package: `oracle.sql.json`

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**

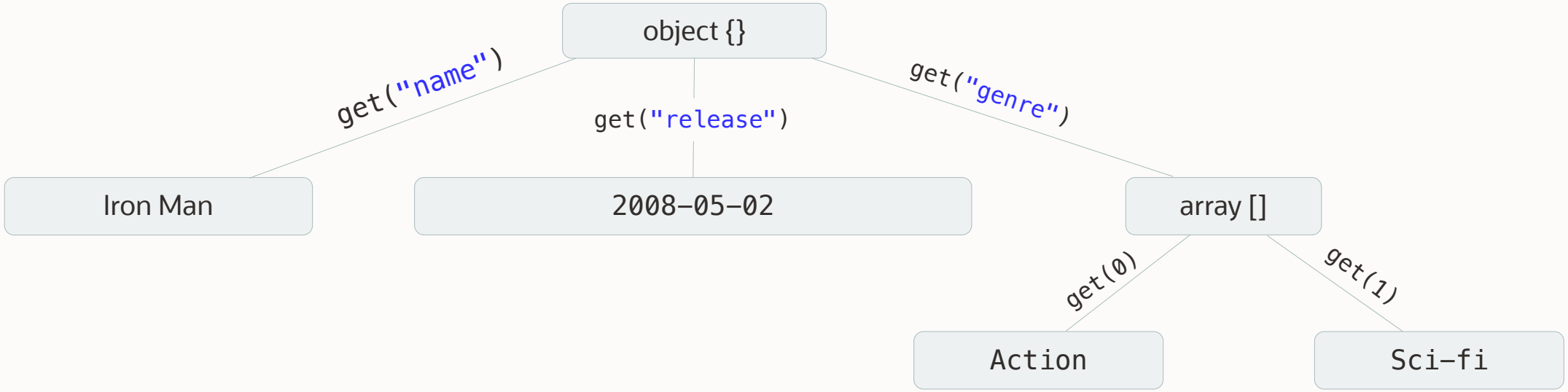
```
createJsonTextGenerator(OutputStream)
createJsonTextGenerator(Writer)
createJsonTextValue(InputStream)
createJsonTextValue(Reader)
createJsonTextParser(InputStream)
createJsonTextParser(Reader)
```

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

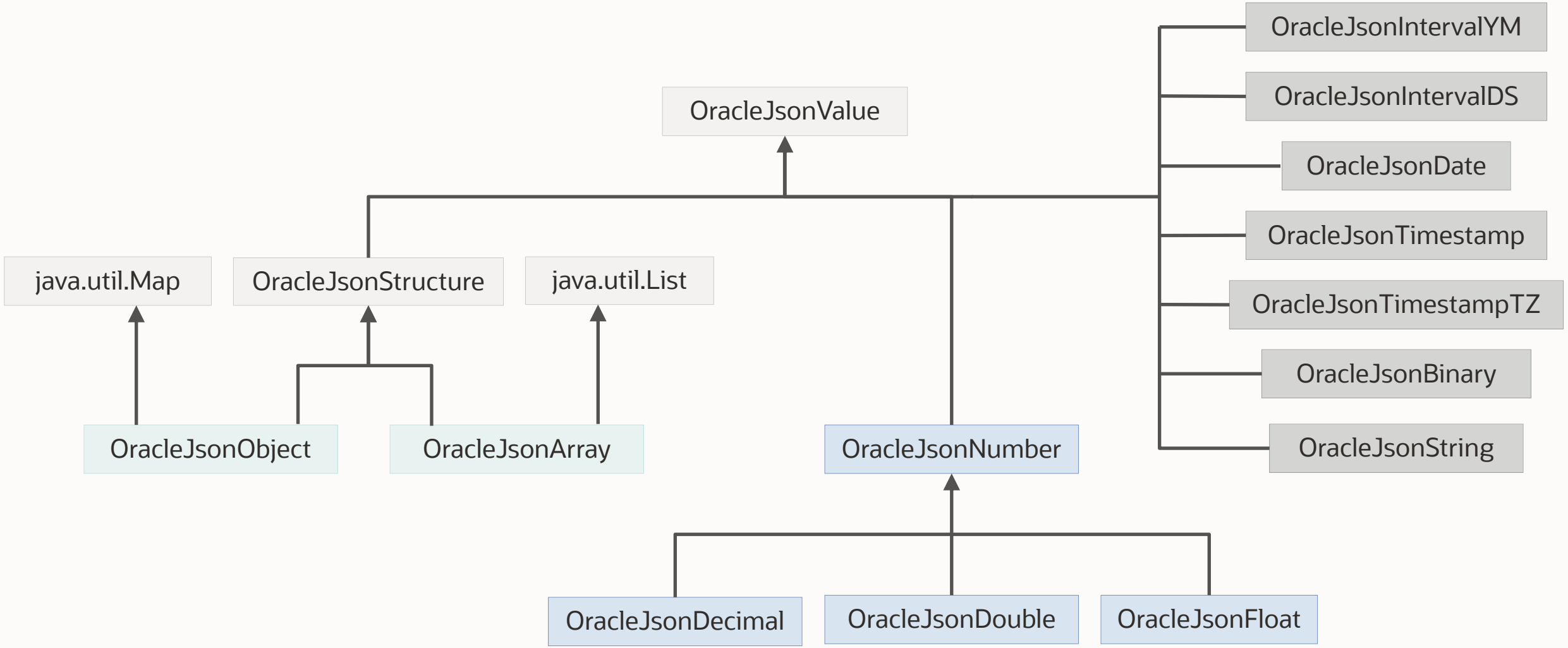
```
OracleJsonFactory factory = new OracleJsonFactory();

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:

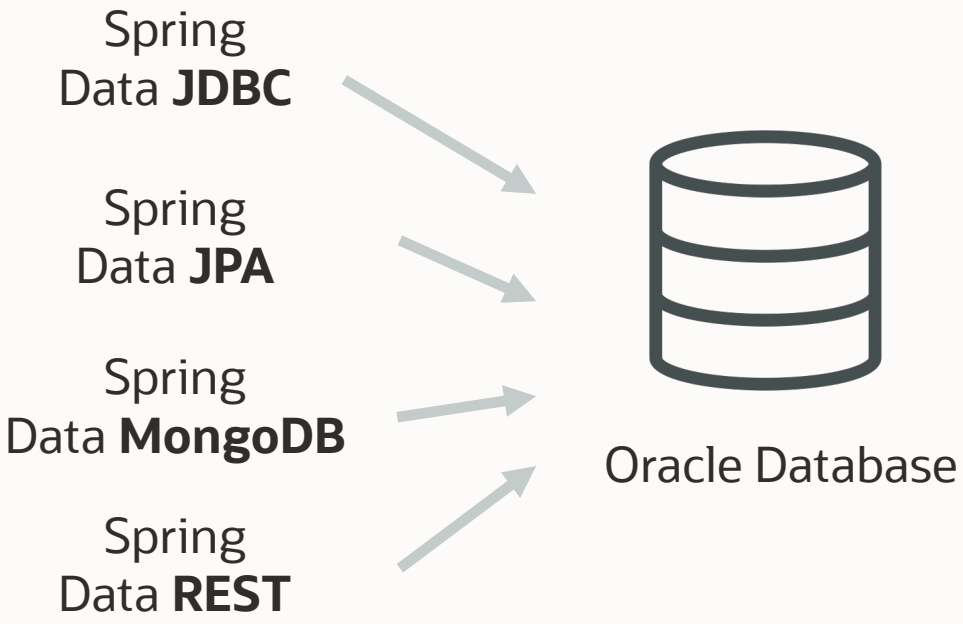
https://github.com/oracle/json-in-db/tree/master/JdbcExamples

Spring Data and JSON

Using Spring Data with JSON storage in Oracle Database

Spring Data

Provides a consistent programming model across different data stores.



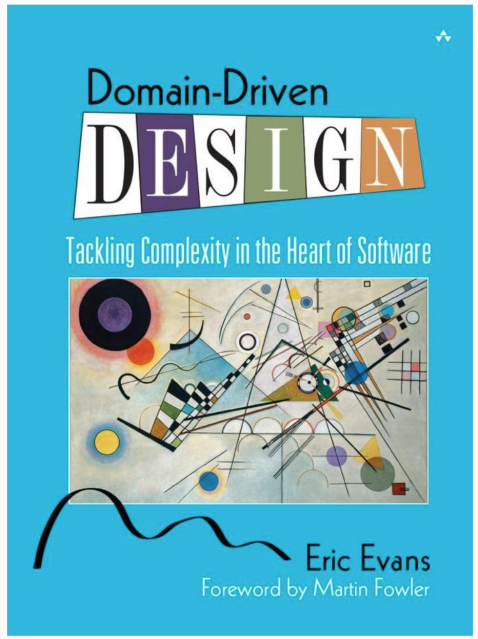
Oracle Database is compatible with multiple different spring data adapters.

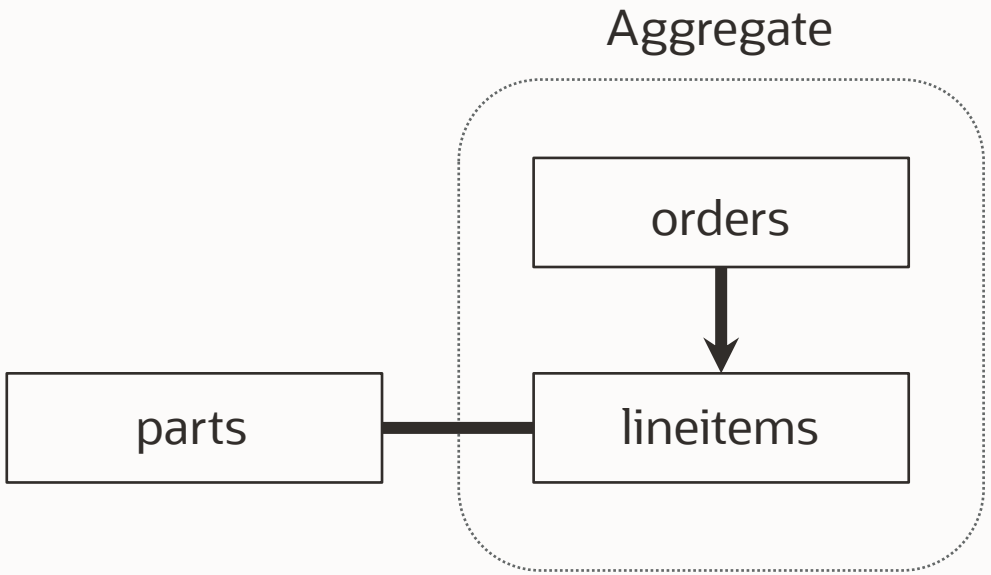
```
public class Movie {
  @Id
  Integer id;
  String name;
  String genre;
}
```

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

*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

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

```
@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

```
@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>

}
```

```
Movie ironMan = orderRepo.findById(1);
```

```
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()

```
@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>

}
```

```
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

Renesas and JSON

About Us

Headquarters
TOYOSU FORESIA, 3-2-24 Toyosu, Koto-ku,
Tokyo 135-0061, Japan

Renesas Electronics is a semiconductor company with an outstanding portfolio of global market-leading products. Our mission is to develop a safer, healthier, greener, and smarter world by providing intelligence to our four focus growth segments that are all vital to our daily lives, meaning our products and solutions are embedded everywhere.

Major Operations
Research, development, design, manufacture, sale, and servicing of semiconductor products

Employees
21,017
(Consolidated, as of December 31, 2022)

Automotive
Highly reliable vehicle control, safe & secure autonomous driving and eco-friendly electric vehicles

Industrial
Lean, flexible and smart industry

IoT
Comfortable, safe and healthy lifestyles through IoT

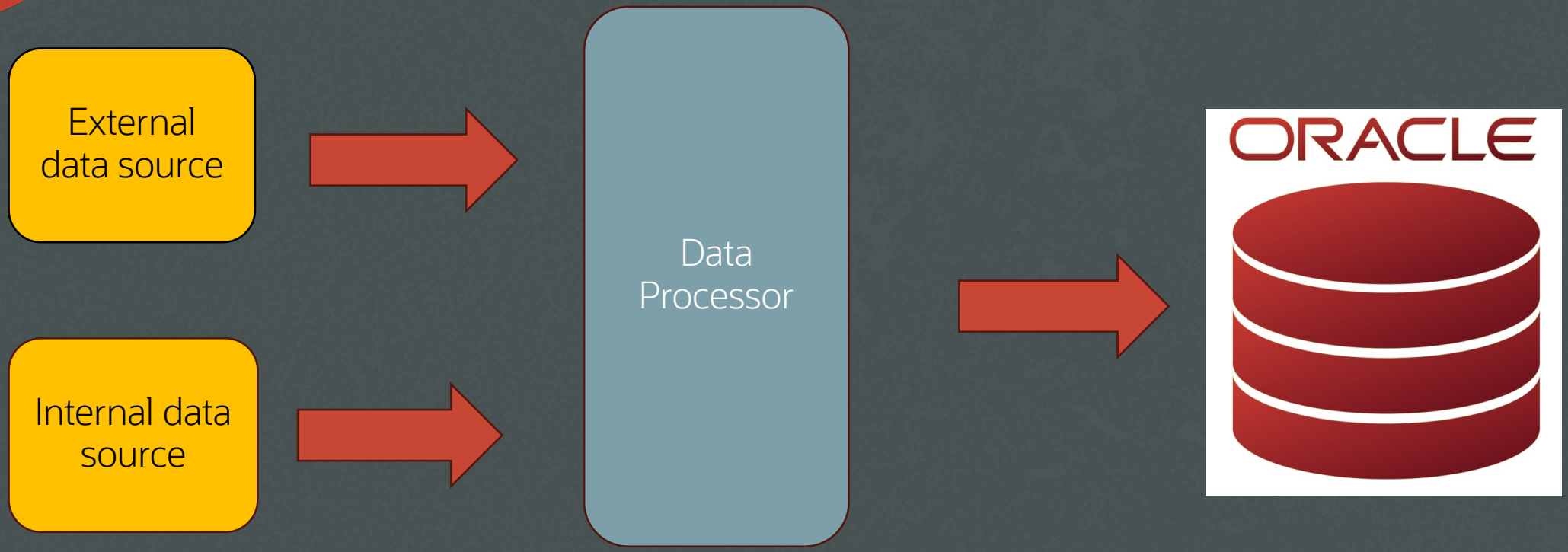
Infrastructure
Robust infrastructure, enabling safety and efficiency

Association with Oracle

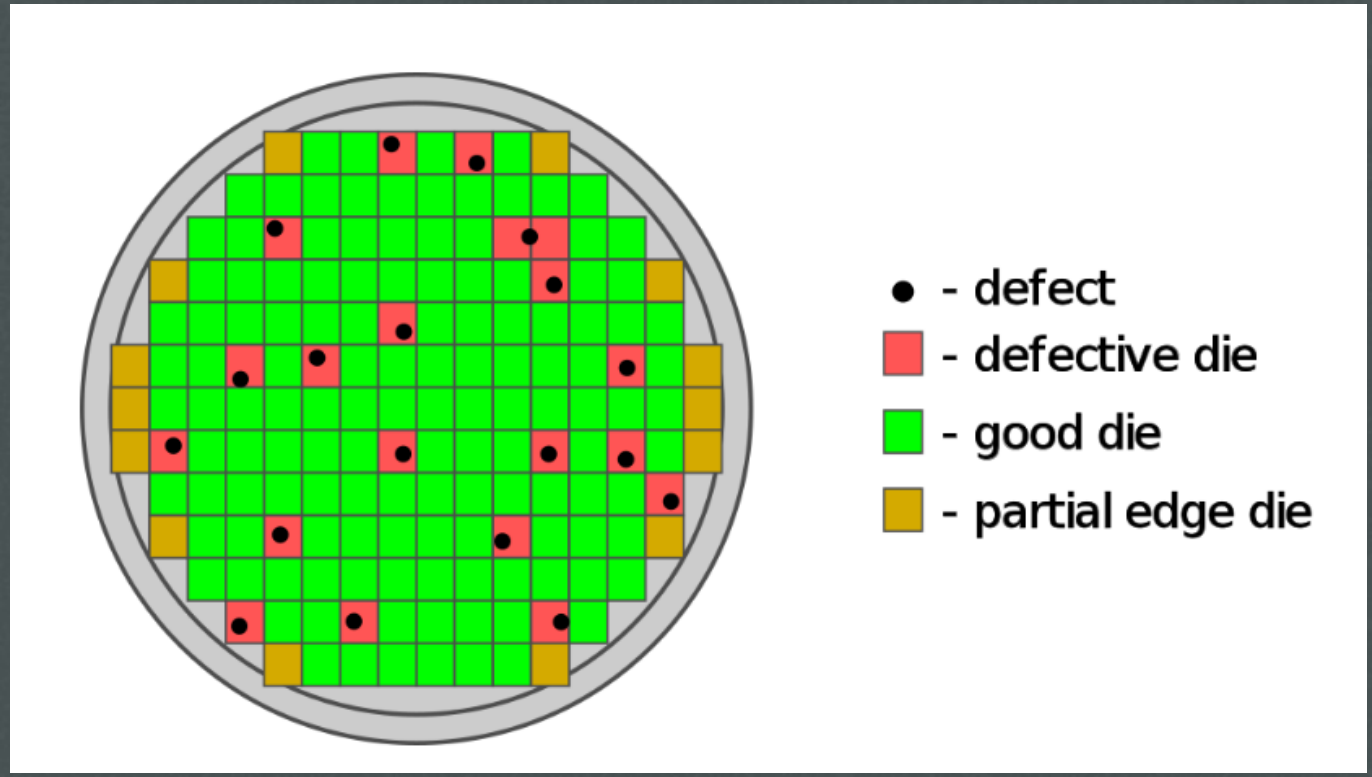
- 20+ years of strong relationship with Oracle
- On-prem to Cloud at Customer
- Cloud at Customer to Oracle Public Cloud

Our use case

Data collection from ATEs (Automated Test Equipment) for yield monitoring is a common practice in the semiconductor world. So, we are constantly focusing on improving the process of data collection, data transfer and data storing.



What are we trying to achieve with JSON?

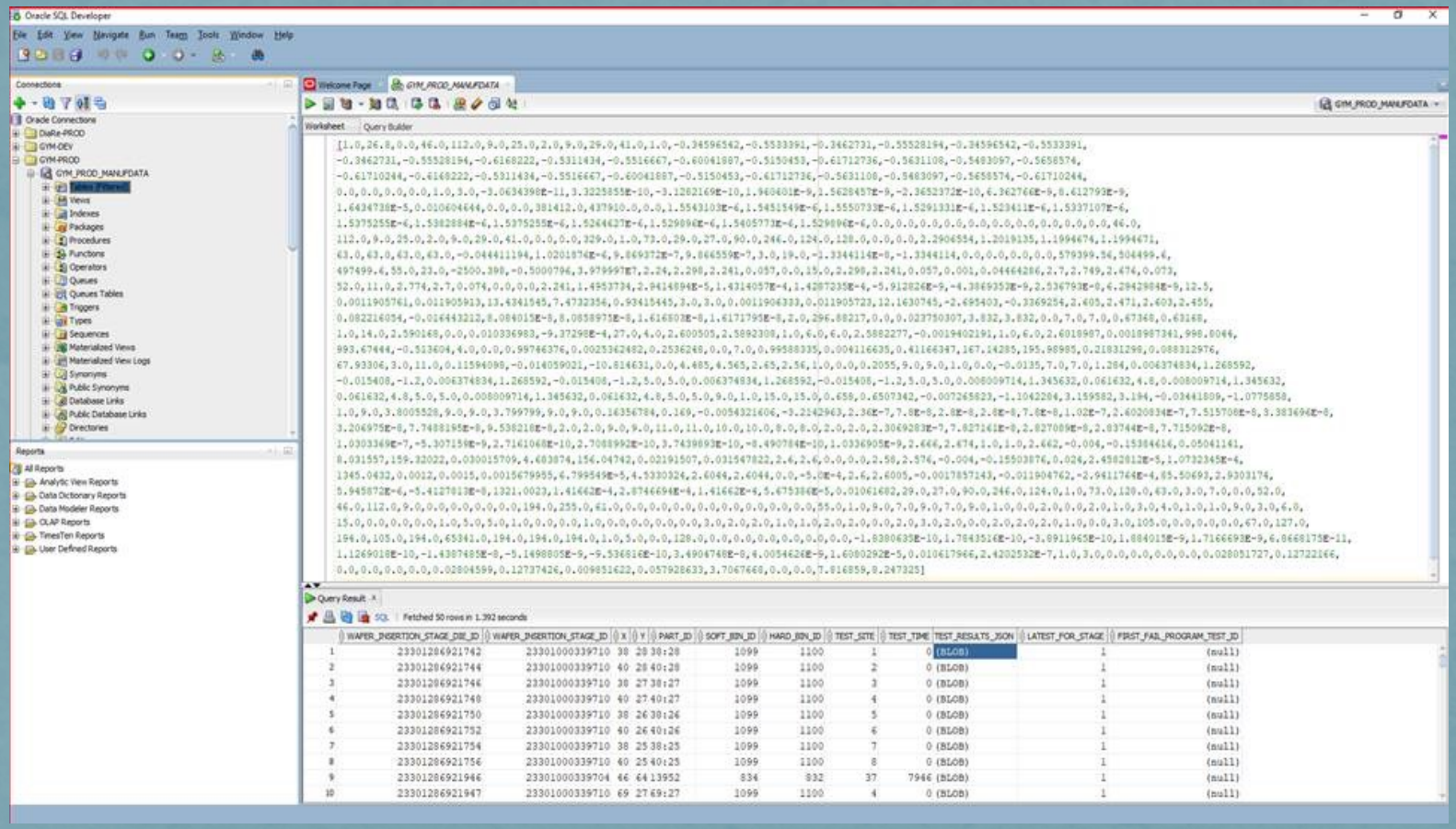


Total number of die: 2000
Total number of tests: 5000
Test result stored in the database: 10 million

Total number of die: 2000
Total number of tests: 5000
Test result stored in the database: 2000

Store data in JSON format

- Storing the data in the database as JSON format with array of numbers.



These numbers are indexed by an index value of the array assigned to the test numbers for test results of each die and the histogram bin number for the histogram data and other applicable scenarios

Data Management for JSON

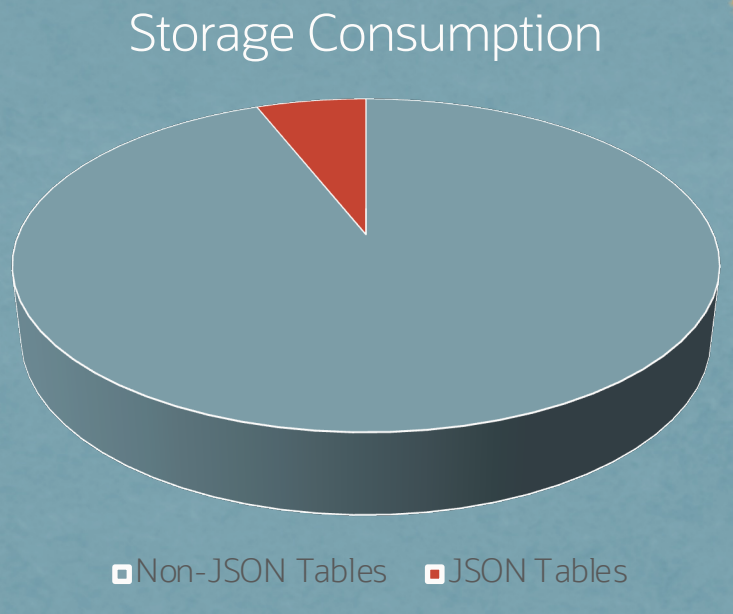
- Our database version 19c on ExaCS – OCI
- We use BLOB type for storing JSON data with check constraint .

```
create table MANUFDATA.LOT_INSERTION_STAGE_TEST_SITE
(
  lot_insertion_stage_test_site_id NUMBER not null,
  lot_insertion_stage_test_id      NUMBER not null,
  test_site                        NUMBER,
  units_tested                     NUMBER,
  units_failed                     NUMBER,
  ....
  ....
  ....
  ....
  histograms_json                  BLOB,
constraint LOT_INSERTION_STAGE_TEST_SITE_PK primary key (LOT_INSERTION_STAGE_TEST_SITE_ID),
constraint LOT_INSERTION_STAGE_TEST_SITE_UK unique (LOT_INSERTION_STAGE_TEST_ID, TEST_SITE),
constraint LOT_INSERTION_STAGE_TEST_SITE_LOT_INSERTION_STAGE_TEST_FK foreign key (LOT_INSERTION_STAGE_TEST_ID) references
MANUFDATA.LOT_INSERTION_STAGE_TEST (LOT_INSERTION_STAGE_TEST_ID),
constraint LISTS_ENSURE_JSON check (histograms_json IS JSON))
LOB (histograms_json) STORE AS SECUREFILE (TABLESPACE MANUFDATA_LOB)
PARTITION BY RANGE (lot_insertion_stage_test_id)
(
partition LOT_INSERTION_STAGE_TEST_SITE_2023011 values less than (230119999999999) TABLESPACE MANUFDATA LOB (histograms_json) STORE AS
LOB_LOT_INSERTION_STAGE_TEST_SITE_2023011(TABLESPACE MANUFDATA_LOB COMPRESS MEDIUM KEEP_DUPLICATES),
partition LOT_INSERTION_STAGE_TEST_SITE_2023012 values less than (230129999999999) TABLESPACE MANUFDATA LOB (histograms_json) STORE AS
LOB_LOT_INSERTION_STAGE_TEST_SITE_2023012(TABLESPACE MANUFDATA_LOB COMPRESS MEDIUM KEEP_DUPLICATES),
partition LOT_INSERTION_STAGE_TEST_SITE_2023013 values less than (230139999999999) TABLESPACE MANUFDATA LOB (histograms_json) STORE AS
LOB_LOT_INSERTION_STAGE_TEST_SITE_2023013(TABLESPACE MANUFDATA_LOB COMPRESS MEDIUM KEEP_DUPLICATES),
.........
.........
.........
partition LOT_INSERTION_STAGE_TEST_SITE_OVER values less than (299999999999999) TABLESPACE MANUFDATA LOB (histograms_json) STORE AS
LOB_LOT_INSERTION_STAGE_TEST_SITE_OVER (TABLESPACE MANUFDATA_LOB COMPRESS MEDIUM KEEP_DUPLICATES)
);
```

- Range Partitioning for the JSON tables
  using advanced compression
- For easy manageability, Retention handling, Pruning , Performance, Hybrid partition tables

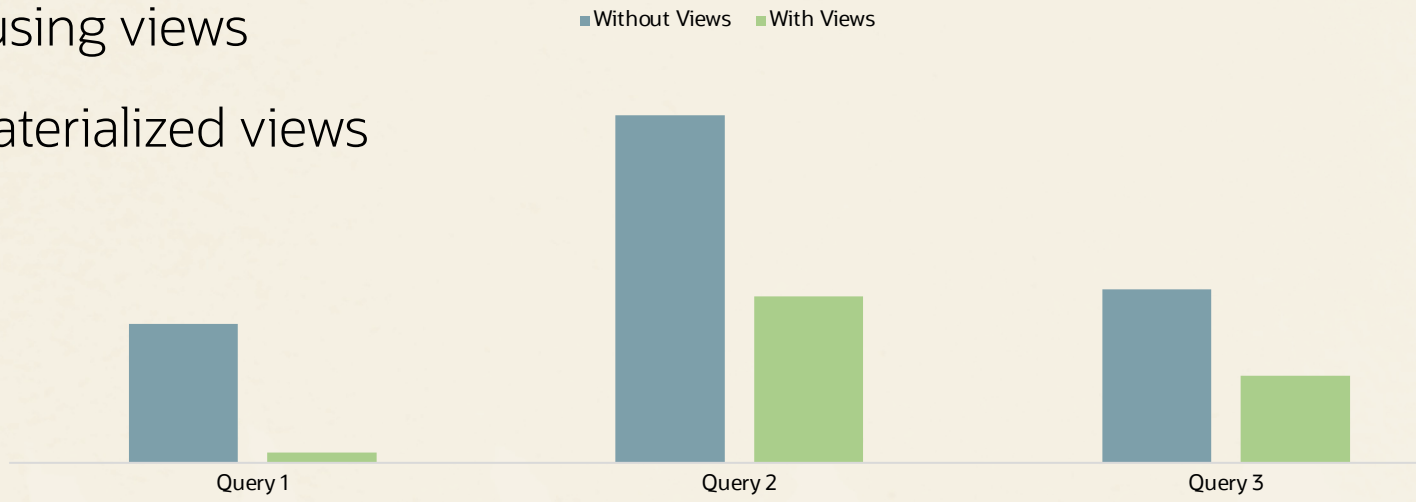
JSON with Advanced Compression

- Stored in SecureFiles LOB Segments
- Storage consumption tested with different compression types
- Advanced Compression – Medium level – Keeping De-duplicates – Best choice
- Good compression ratio (at least 10 to 15 times compared to non-JSON tables stored earlier)
- Modest CPU usage



Performance Boost using views

- Terabytes of JSON data
- Complex queries
- Querying the JSON using inbuilt functions like JSON_TABLE for retrieval
- Documented behavior creating a user defined views boost the query performance by factor 2 times at least
- Change in execution plan using views
- Additional option to use materialized views



ORACLE
CloudWorld

Thank you

Moumita Bardhan,
Senior Manager,
Production Support Systems,
Information Systems Division
Renesas Electronics Corporation

Sriram Ganapathy,
Team Lead Oracle DBA and Data Analysis
Production Support Systems,
Information Systems Division
Renesas Electronics Corporation

Join us on Slack

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

#oracle-db-json

Session Survey

Best Practices for Modern Application Development Using JSON and Java

LRN2977

Appendix

Oracle Performs Like a NoSQL database

Elastic compute and storage

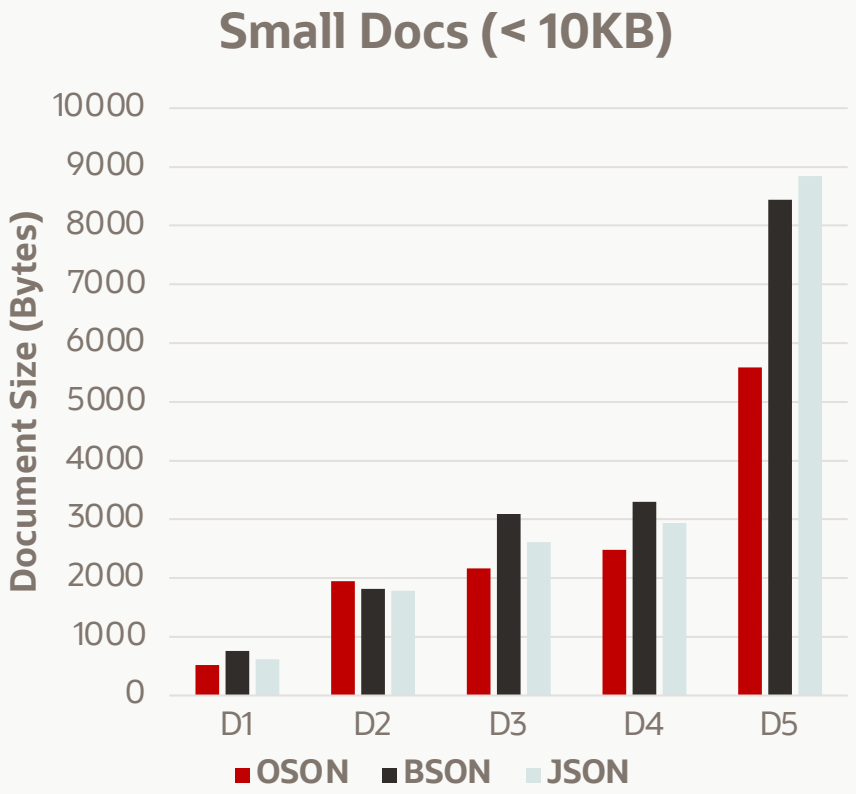
Single-digit latency reads and writes

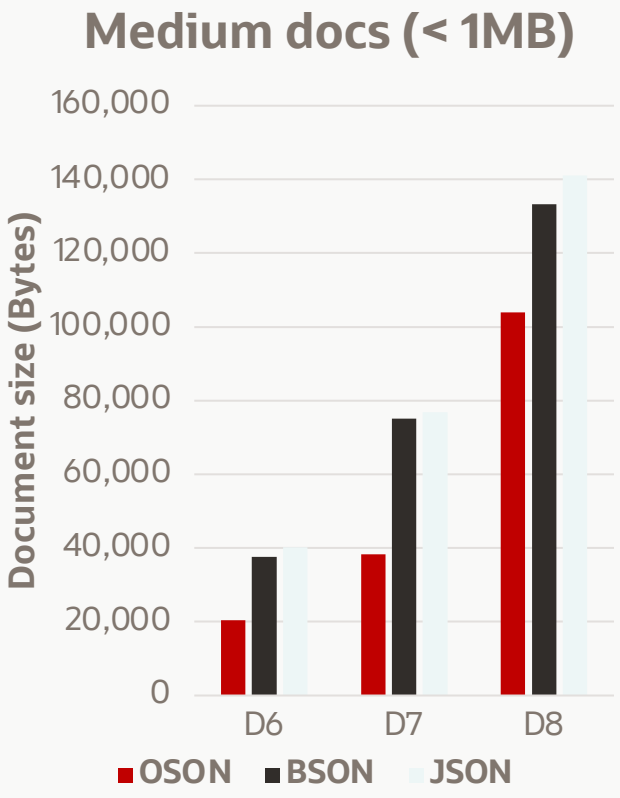
Highly available

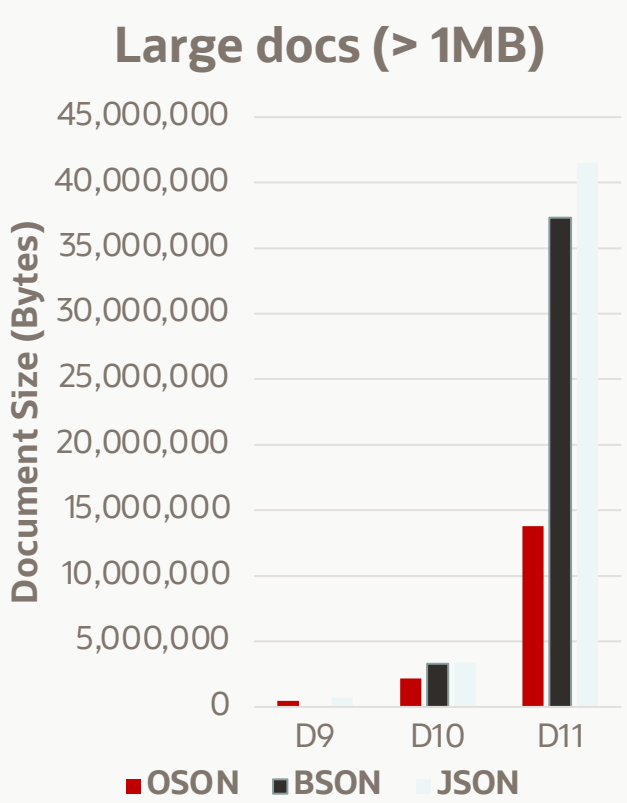
Low-cost

OSON Compression

Up to 2.7x more compact







Tree Model Sources

`OracleJsonObject` and **`OracleJsonArray`** support updates when not directly created from OSON.

Description	Source	Read-only?
From the database	ResultSet.getObject()	Yes
Created from OSON	createJsonBinaryValue(ByteBuffer) createJsonBinaryValue(InputStream)	Yes
Created from JSON text	createJsonTextValue(Reader) createJsonTextValue(InputStream)	No
Created programmatically	createObject() createArray()	No
Copied	createObject(OracleJsonObject) createArray(OracleJsonArray)	No

JSR-374 JSON-P

- JSON-P (JSON Processing) is a Java API to parse and generate JSON
- `oracle.sql.json` can loosely be thought of as an extension of JSON-P
- JSON-P 1.0 -> **`javax.json`**
- JSON-P 2.0 -> **`jakarta.json`**
- Has tree model and event model
- Differences from `oracle.sql.json`:
  - supports extended types
  - supports a mutable tree model
  - supports conversion to and from OSON
  - does not support builders/reader or the services API

JSON-P: Wrap and Unwrap

- Interfaces are different views of the same data
- Wrap and unwrap do not make a deep copy or incur a performance penalty (e.g. in-place supported for javax.json)
- `javax.json` instances produced by JDCB implement `java.sql.Wrapper`

wrap()

oracle.sql.json	jakarta.json
OracleJsonObject	JsonObject
OracleJsonArray	JsonArray
OracleJsonDecimal	JsonNumber
OracleJsonDouble	
OracleJsonFloat	
OracleJsonString	JsonString
OracleJsonBinary	
OracleJsonTimestamp	
OracleJsonDate	
OracleJsonIntervalDS	
OracleJsonIntervalYM	
OracleJsonGenerator	JsonGenerator
OracleJsonParser	JsonParser

Wrapper.unwrap()

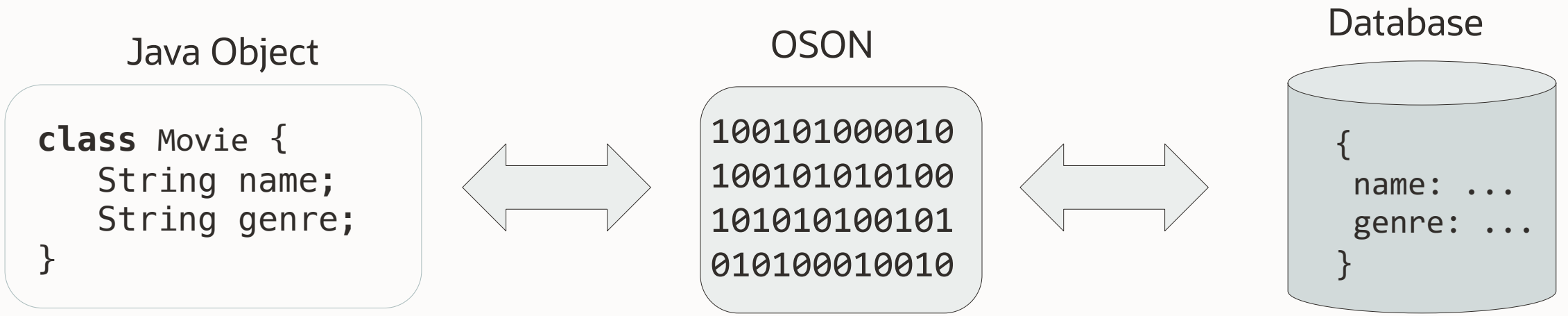
When to use JSON-P?

- Your application already implements `jakarta.json` interfaces and you want the database to consume them
- Your application uses a third-party library that implements `jakarta.json` and you need the database to consume them
- You want to keep your application independent of Oracle Database specific APIs and extended types
- You want to use JSON-B

```
ResultSet rs = stmt.executeQuery("SELECT data FROM movie ");
rs.next();
JsonObject ironMan = rs.getObject(1, jakarta.json.JsonObject.class);
```

JSR-364 JSON-B

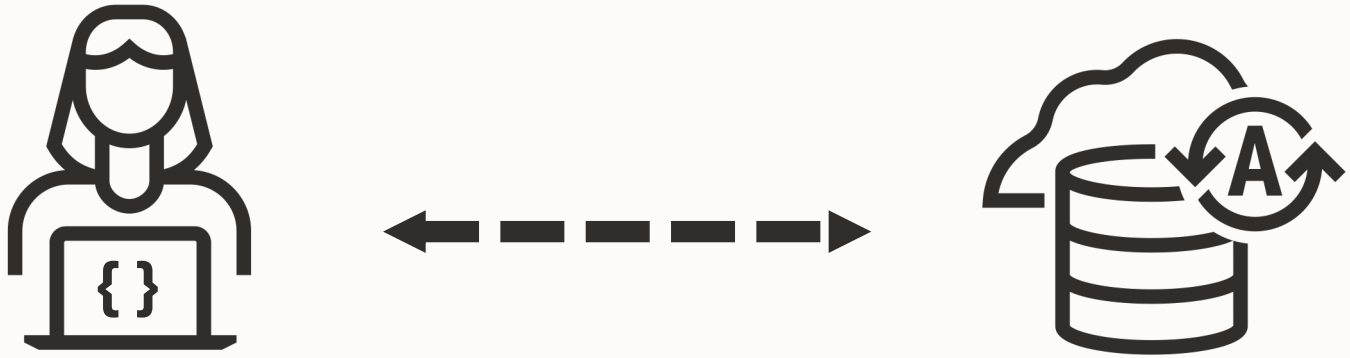
- **JSON-B** (JSON Binding) convert Java objects to/from JSON
  - Default mapping between any Java object and JSON
  - Java Annotations to customize default mapping if necessary
- **Yasson** is the default implementation of JSON-B
- Use **Yasson** and **JDBC** to persist Java objects as binary JSON ***without parsing/serializing JSON text***
- Provides efficient ingest, retrieve, and query of application domain objects
- Complex objects with nested relationships can be retrieved without joins



JSON Collections

NoSQL-style access to JSON in Oracle Database

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

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();
```

SQL – but only when you need it...

JSON Collections

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

Simple, flexible persistence for applications, microservices

SQL/JSON

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

Powerful analytics and reporting directly over collections