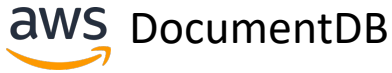
ORACLE

Oracle Database JSON Workshop Preview

Josh Spiegel

Software Architect, Oracle Database

Oracle Database as a JSON Document Store



- A type of NoSQL database
- Low-latency, high throughput
- Store collections of JSON objects
  - Simple
  - Flexible
  - App friendly
- Low-cost

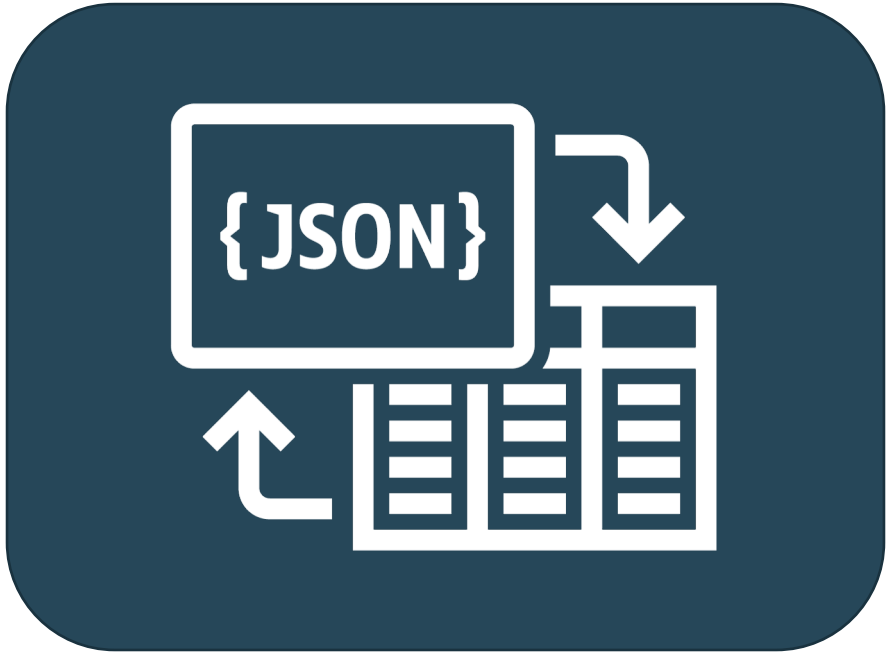
```
{
  "movie_id" : 1652,
  "title" : "Iron Man 2",
  "date" : "2010-05-07",
  "cast" : [
    "Robert Downey Jr.",
    "Larry Ellison"
  ]
}
```

Today's JSON Workshop

Introduction
- Benefits of JSON document databases
- Oracle Database as a document store

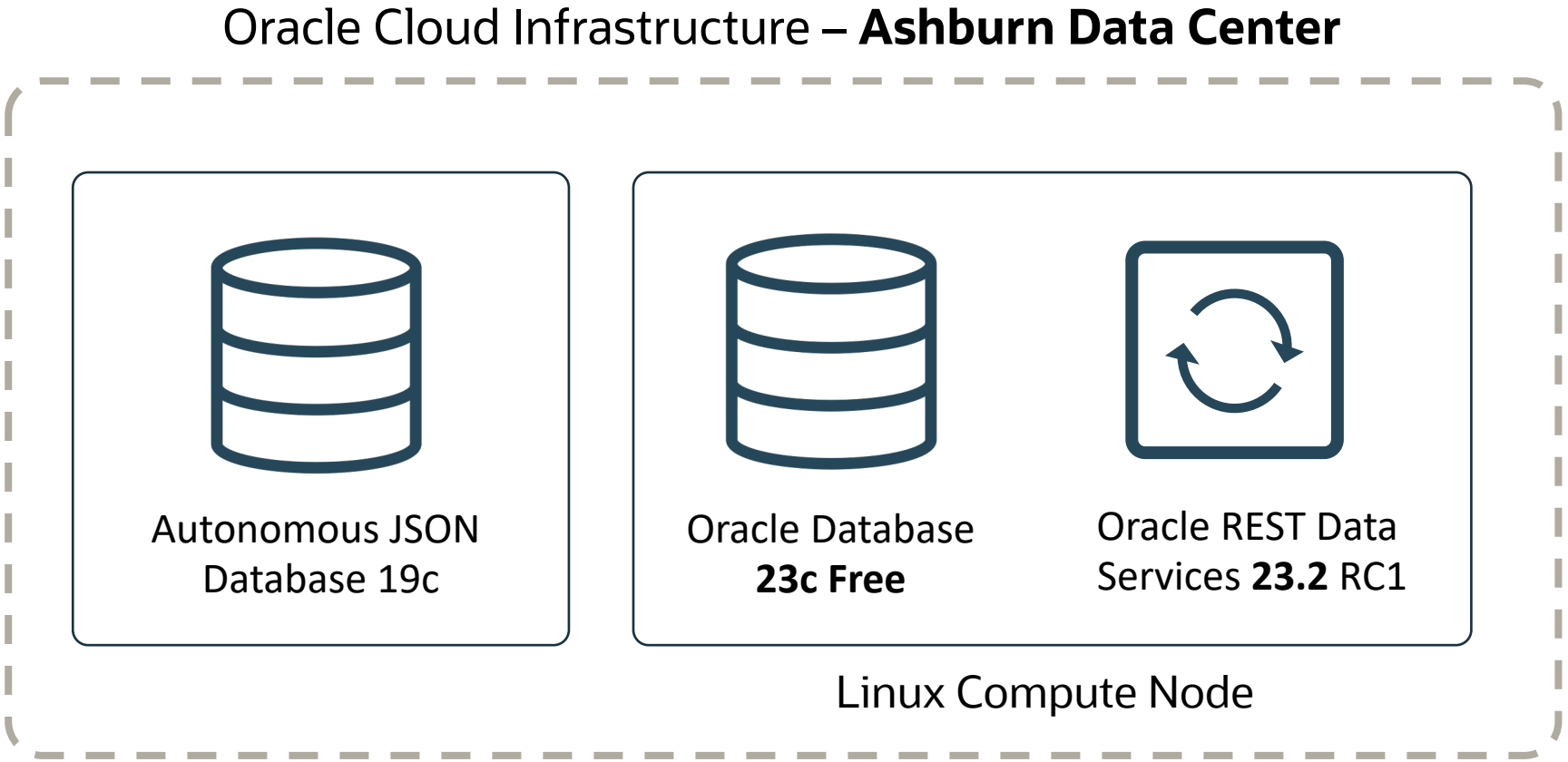
SQL/JSON
- JSON Storage
- Query and update JSON
- Load JSON from files

JSON Collections
- Underlying data model
- Connecting Mongo DB clients to Oracle Database
- Migrating data from MongoDB to Oracle



First demo of new and unreleased features!

Demonstration



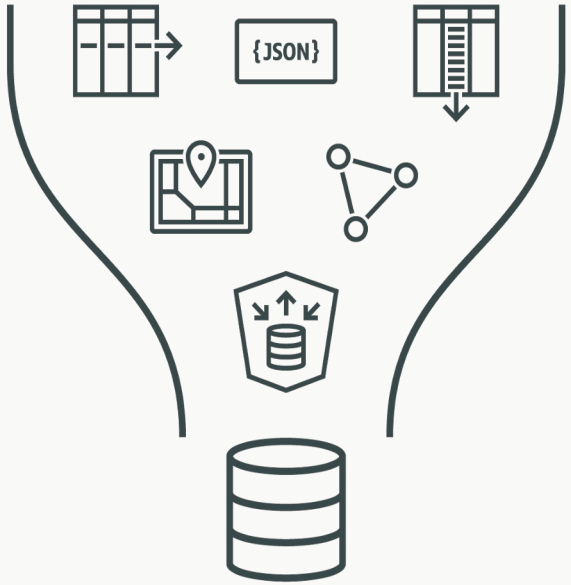
ORACLE

Oracle Database JSON

Josh Spiegel
Software Architect, Oracle Database

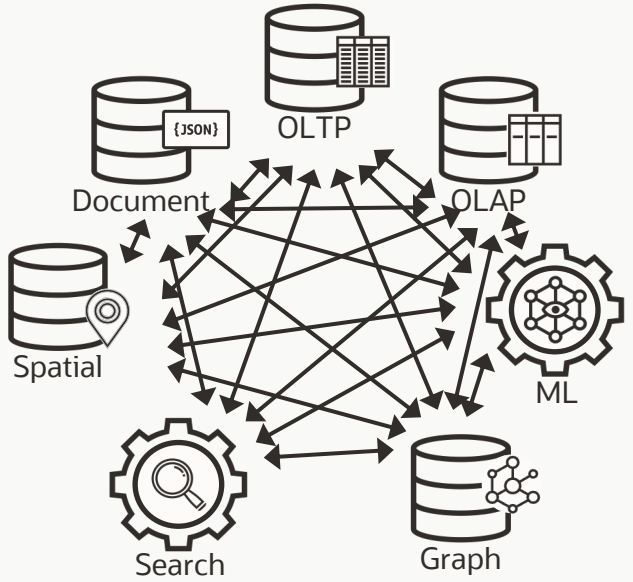
Oracle Converged Database

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 vs Multi-Model

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

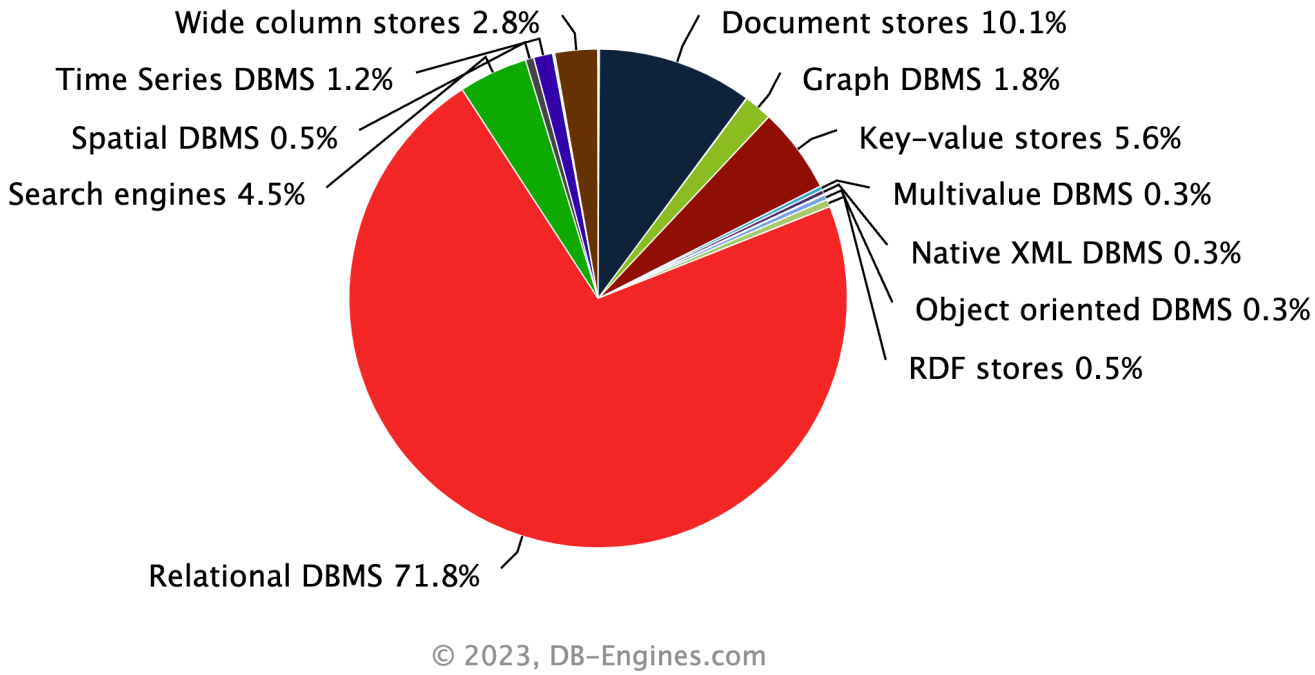
Multi-Model SQL

Database Popularity 2023 (db-engines.com)

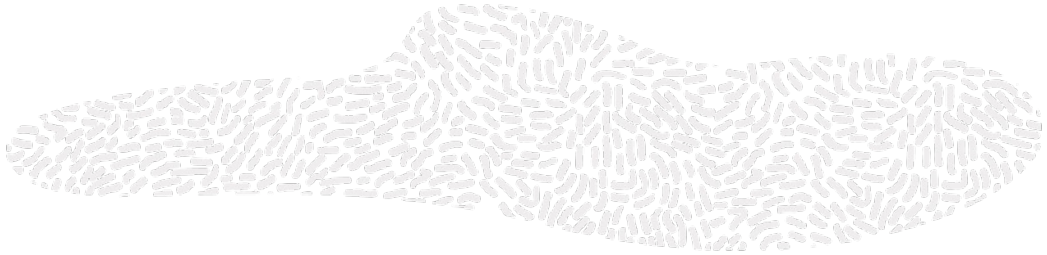
Top-10 Ranked Databases

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



NoSQL JSON Document Stores



MongoDB. Couchbase CosmosDB

aws DocumentDB Google Firestore

- Store collections of JSON (JavaScript Object Notation)
- A simple, self-describing, flexible way to model objects
- String, numbers, true/false, Nested repeating values

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

Why store JSON in a database?

```
{
  "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() {
   ...
}
```

Schema-flexible

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

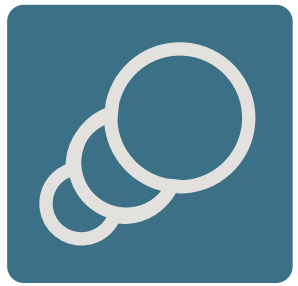
Less Impedance Mismatch

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

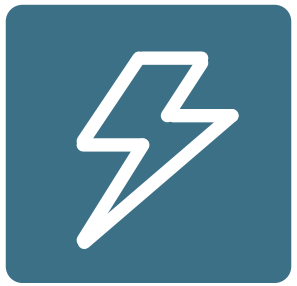
Data Exchange

- One representation across tiers
- Human readable
- REST, JSON are ubiquitous

NoSQL Document Store Features



Elastic compute and storage



Single-digit latency reads and writes

Highly available

Low-cost

NoSQL Document Store - Data as code

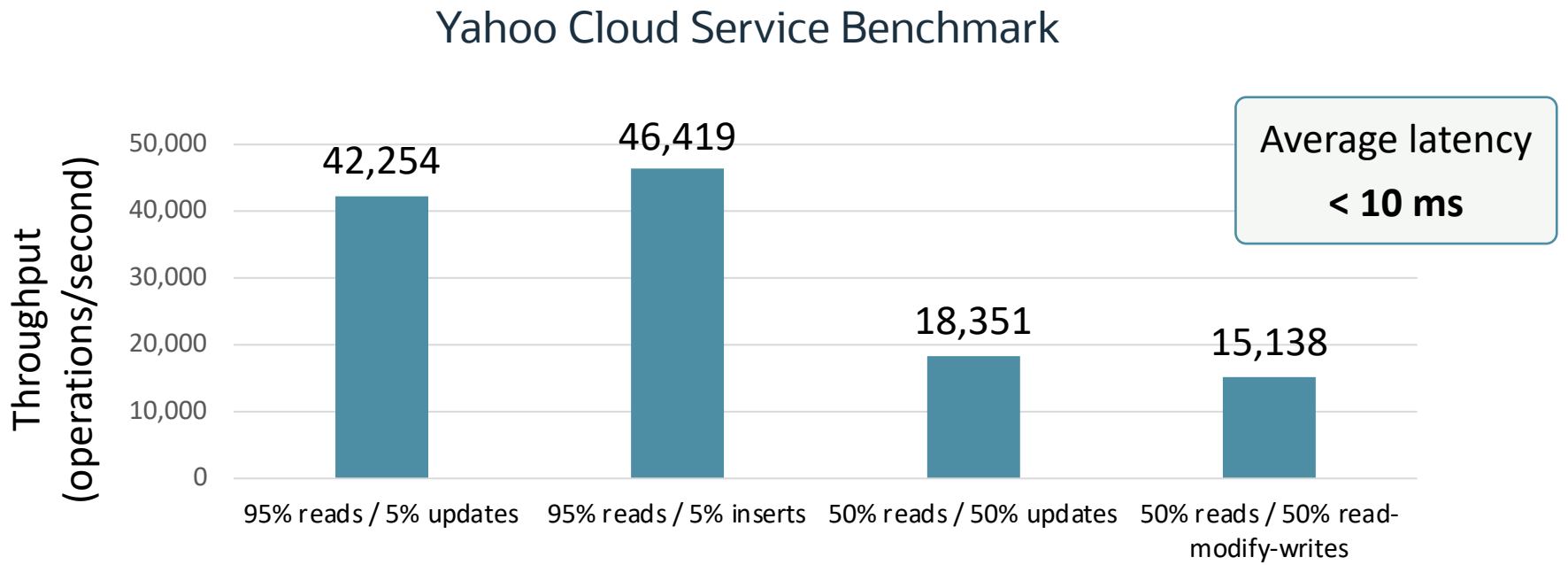
NoSQL Access

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

db.movies.find({"_id" : 123});
```

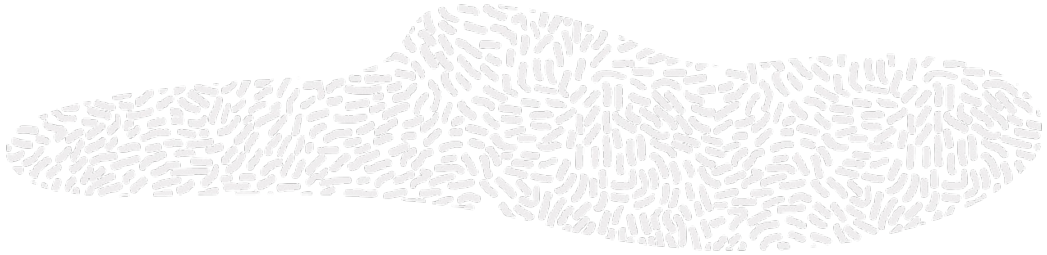
SQL Access

```
PreparedStatement ins =
  con.prepareStatement( "INSERT INTO movies VALUES (:1,:2:)" );

ins.setNumber(1, id)
ins.setString(2, movieName);
ins.execute();

ResultSet rs =
   stmt.executeQuery( "SELECT * FROM movies WHERE id = 123" )
```

Oracle Database: all the features of a single-purpose JSON document store – with superior performance



** Autonomous JSON Database with 8 OCPU running in San Jose region, Mongo API



But without losing all the strengths of Oracle Database when you store JSON…

- 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

Autonomous Database offers workload-optimized options

OLTP+ JSON workloads

Autonomous Database for Transaction Processing and mixed workloads - ATP

Pre-configured for row format, indexes, and data caching to accelerate **transaction processing** and **mixed workloads**

- **Converged database** with no data storage limits (relational or JSON), JSON data fully supported
- Includes **Oracle Database API for MongoDB**

JSON-centric workloads

- Autonomous JSON Database – AJD

Price optimized for transactions and analytics on **JSON data**

- Same features as ATP but **75% lower price**
- **Unlimited JSON collections** + up to 20GB non-JSON data
- Includes **Oracle Database API for MongoDB**
- Great option for a "start small and grow big" app dev strategy—can be **upgraded to ATP with a click of a button**

Single-click upgrade to ATP

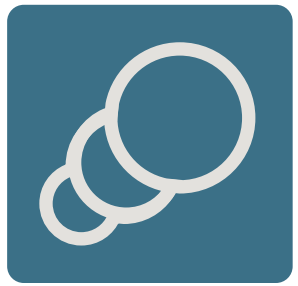
Analytics workloads including JSON data

- Autonomous Database for analytics and data warehousing – ADW

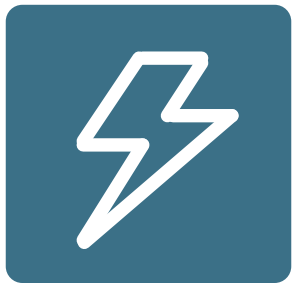
Pre-configured for columnar format, partitioning, and large joins to accelerate **analytics, data warehouse**, and **data lakehouse**

- Same features as ATP, optimized for analytics
- JSON data fully supported
- Includes **Oracle Database API for MongoDB**

Autonomous JSON Database



Elastic compute and storage



Single-digit latency reads and writes

Highly available

Low-price, always-free tier

Accessing JSON with SQL

Traditional Relational Model

- A schema contains tables
- A table contains rows
- A table is flat
- Rows are structured
- Data is accessed with SQL
- Related rows are joined

Flat, structured tables

id	title	date
123	Iron Man	2010-05-07
345	Thor	2022-07-08

Accessed with SQL

```
SELECT m.title, m.date
FROM   movies m
WHERE  m.id = 123
```

Relational Model + JSON Columns

- Schema-flexible JSON stored within a structured column
- SQL extended to process JSON column values
- Stored using query-efficient OSON binary format

```
CREATE TABLE movies (data JSON);

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

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

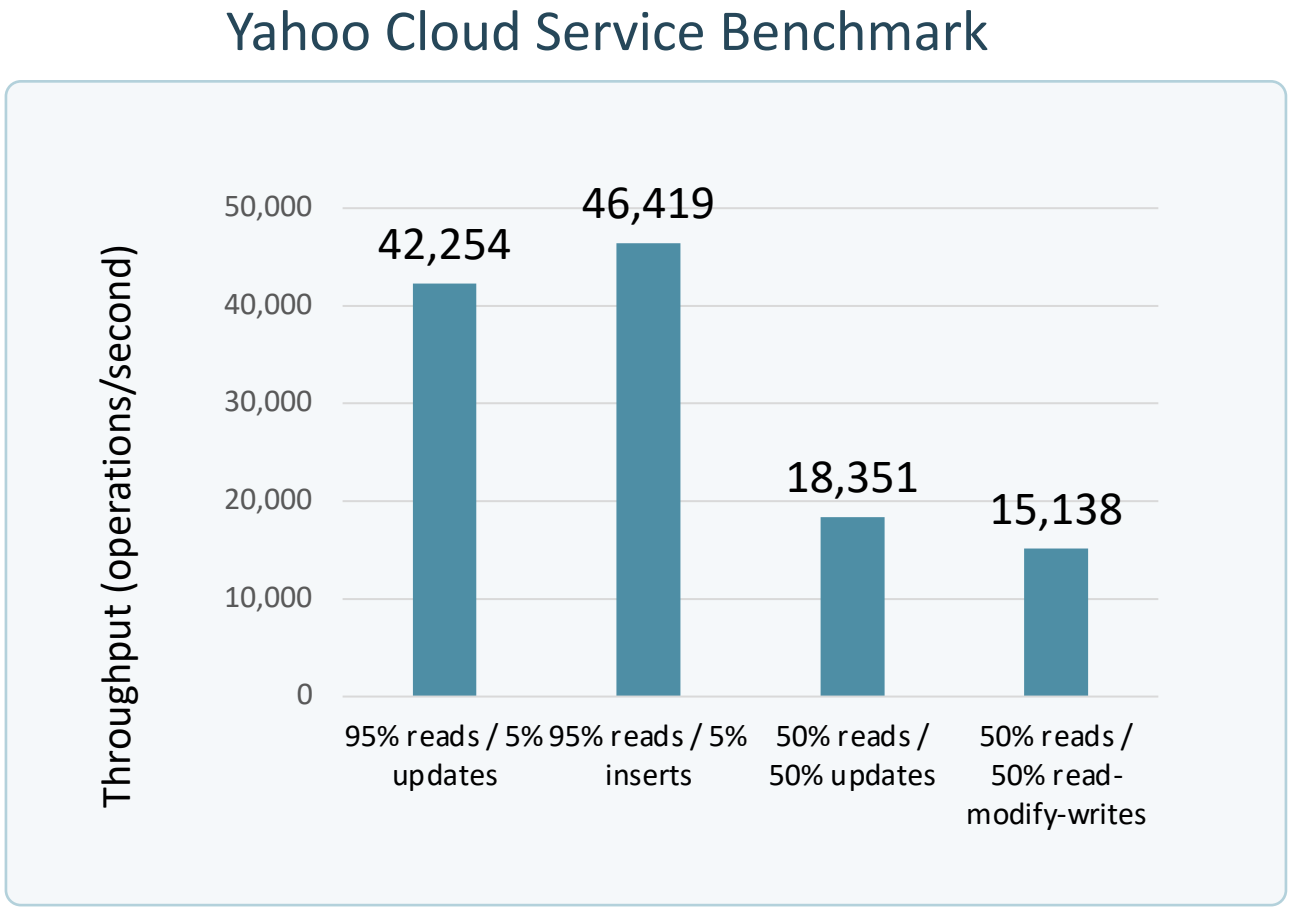
- Standard
  - OBJECT { }
  - ARRAY [ ]
  - STRING
  - TRUE/FALSE
  - NULL
  - NUMBER
- Extended
  - BINARY_FLOAT
  - BINARY_DOUBLE
  - TIMESTAMP/DATE
  - INTERVALDS/INTERVALYM
  - RAW

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

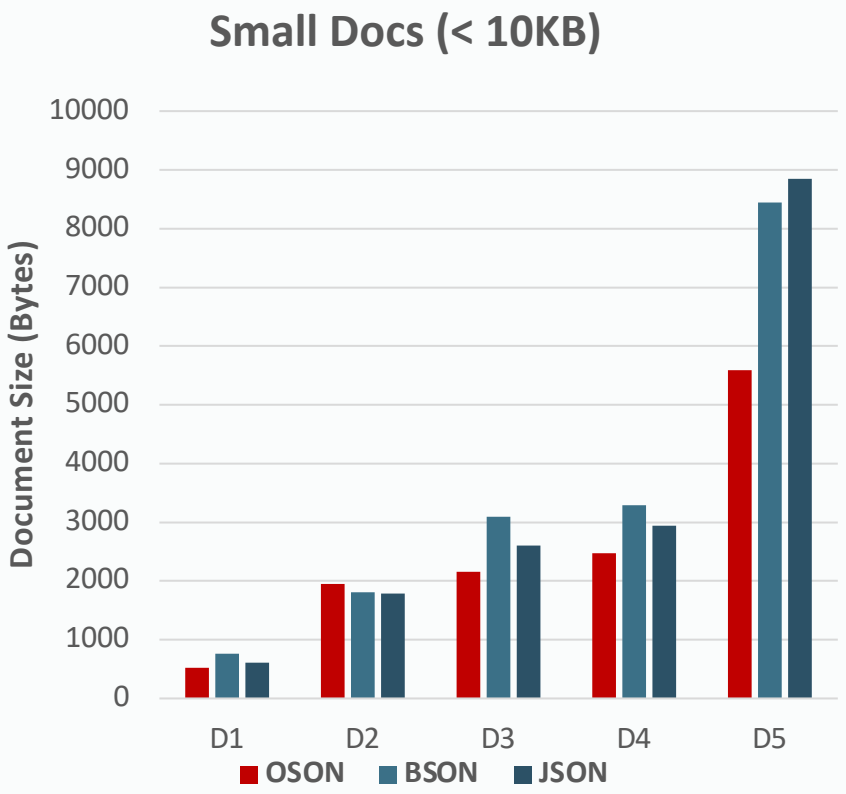
- Faster path evaluation
- Efficient random access
- Smaller than JSON text and BSON
- Less network and storage IO
- No text parsing or serialization

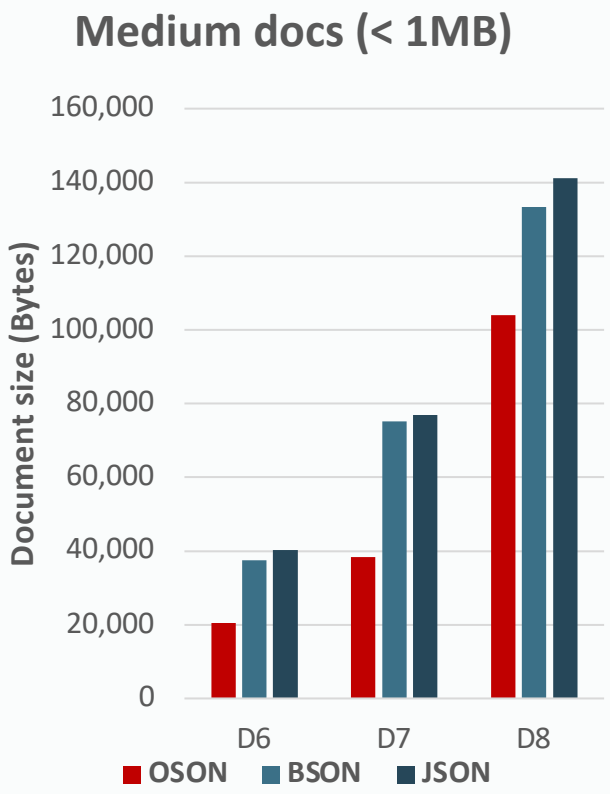


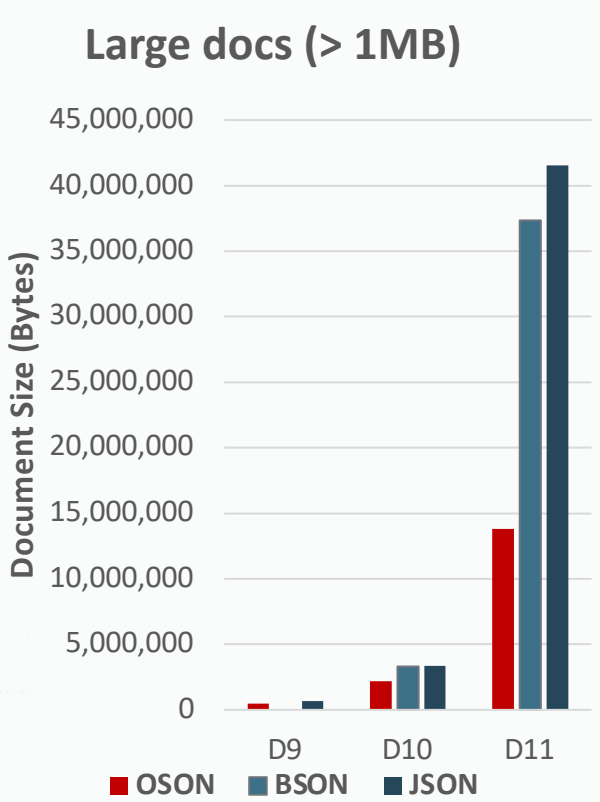
See VLDB 2020:
Native JSON Datatype Support: Maturing SQL and NoSQL convergence in Oracle Database

OSON Compression

Up to 2.7x more compact

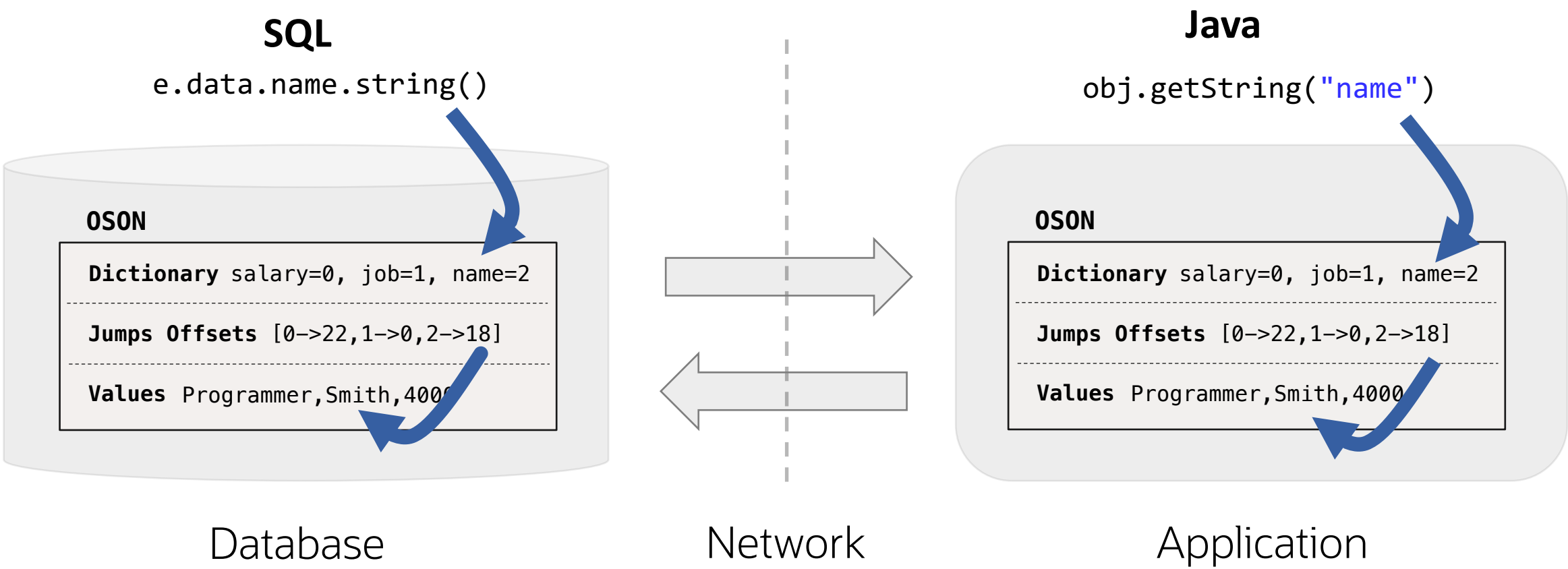






OSON Performance

```
{"name":"Smith","job":"Programmer","salary":40000}
```



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

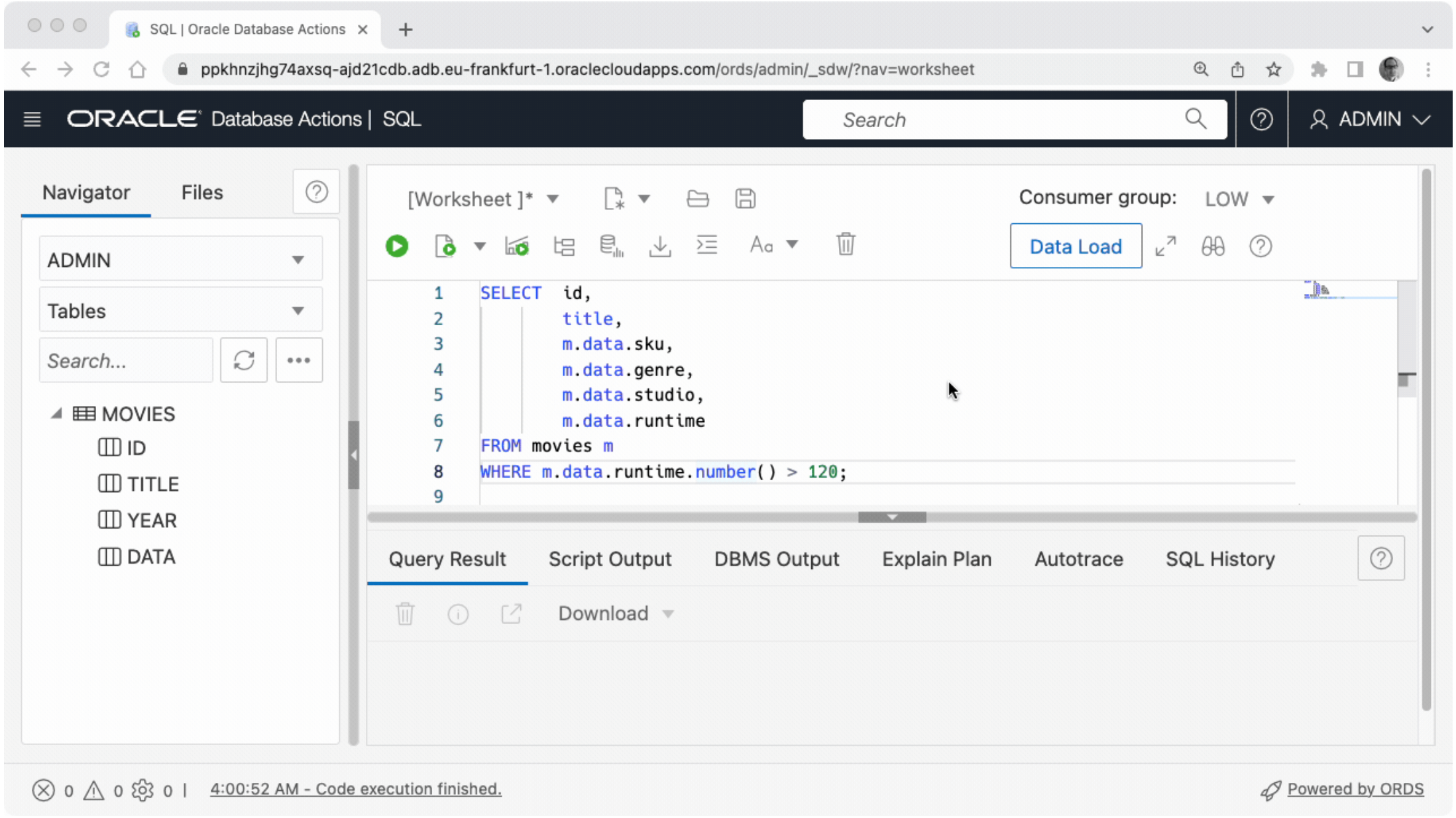
```
SELECT title, actor
FROM movies NESTED jcol COLUMNS (
  title,
  NESTED actors[*] COLUMNS (
    actor
  )
)
WHERE id = 123
```


Simple Dot Notation

```
CREATE TABLE movies(
        id      number PRIMARY KEY,
        title   varchar2(50),
        year    number,
        data    JSON);


INSERT INTO movies VALUES( 1, 'Avatar', 2009,
  '{"sku": "LYG56160", "runtime": 162,
   "cast": ["Sam Worthington", "Zoe Saldana"},
   "studio": ["20th Century Studios",
              "Lightstorm Entertainment"} '
);
```

```
SELECT m.id,
       m.title,
       m.data.sku,
       m.data.genre,
       m.data.studio
FROM movies m
WHERE m.data.runtime > 120;
```

Select JSON attributes as if they were columns.



JSON_QUERY

```
CREATE TABLE movies(
        id      number PRIMARY KEY,
        title   varchar2(50),
        year    number,
        data    JSON);


INSERT INTO movies VALUES( 1, 'Avatar', 2009,
  '{"sku": "LYG56160", "runtime": 162,
   "cast": ["Sam Worthington", "Zoe Saldana"},
   "studio": ["20th Century Studios",
              "Lightstorm Entertainment"} '
);
```

```
SELECT
   id,
   title,
   m.data.sku,
   JSON_QUERY(
      m.data,
      '$.genre'
      EMPTY ARRAY ON EMPTY
   ),
   m.data.studio
FROM movies m
WHERE m.data.runtime > 120;
```

JSON_QUERY to have more control over how a JSON value read.

JSON_TABLE: map repeating values to rows

```
{
  "movie_id": 200,
  "title": "Alien",
  "sku": "RSD56032",
  "year": 1979,
  "runtime": 117,
  "crew": [
     {"job": "producer",
      "names": ["David Giler",
                "Gordon Carroll"]
     },
     {"job": "director",
      "names": ["Ridley Scott"]
     },
     {"job": "screenwriter",
      "names": ["Dan O'Bannon",
                "Ronald Shusett"]
     }
  ]
}
```

```
select jt.*
from movies m, JSON_TABLE(m.data, '$' columns (
 "ID" path '$.movie_id',
  title,
  year NUMBER,
  nested path '$.crew[*]' columns (
  job,
  nested path '$.names[*]' columns("NAME" path '$'))))
jt;
```

```
ID  TITLE YEAR JOB          NAME
--  ----- ---- ----------- --------------
200 Alien 1979 producer     David Giler
200 Alien 1979 producer     Gordon Carroll
200 Alien 1979 director     Ridley Scott
200 Alien 1979 screenwriter Dan O'Bannon
200 Alien 1979 screenwriter Ronald Shusett
```


NESTED is a shortcut for left outer with JSON_TABLE

```
{
  "movie_id": 200,
  "title": "Alien",
  "sku": "RSD56032",
  "year": 1979,
  "runtime": 117,
  "crew": [
     {"job": "producer",
      "names": ["David Giler",
                "Gordon Carroll"]
     },
     {"job": "director",
      "names": ["Ridley Scott"]
     },
     {"job": "screenwriter",
      "names": ["Dan O'Bannon",
                "Ronald Shusett"]
     }
  ]
}
```

```
select jt.*
from movies nested data columns (
  "ID" path '$.movie_id',
  title,
  year NUMBER,
  nested path '$.crew[*]' columns (
  job,
  nested path '$.names[*]' columns("NAME" path '$'))))
jt;
```

```
ID  TITLE YEAR JOB          NAME
--  ----- ---- ---------- -------------

200 Alien 1979 producer     David Giler
200 Alien 1979 producer     Gordon Carroll
200 Alien 1979 director     Ridley Scott
200 Alien 1979 screenwriter Dan O'Bannon
200 Alien 1979 screenwriter Ronald Shusett
```


JSON Generation: "from Tables to JSON Data"

Table "customers"

ID	FIRST	LAST	COUNTRY
11	Eliana	Carillo	US
12	Keiran	Stanton	EN
13	Shanice	Collins	US

Table "watches"

cust_id	movie_id	type
11	1	Subscription
11	2	Subscription
12	1	Purchase
13	2	Trial

Table "movie"

ID	TITLE	YEAR	DATA
1	Avatar	2009	{..}
2	Ghostbusters II	1989	{..}

```
select JSON {'name'   : c.first || ' ' || c.last,
             'movies': [select m. title
                        from movies m, watched w
                        where w.movie_id = m.id
                        and w.cust_id = c.id]
            }
from customers c;
```

```
{
  "name":"Eliana Carrilo",
  "movies":["Avatar","Ghostbusters II"]
}

{
  "name":"Keiran Stanton","movies":["Avatar"]
}

{
  "name":"Shanice Collins","
   movies":["Ghostbusters II"]
}
```

Automatically Map JSON to Rows

```
SELECT JSON_DATAGUIDE(movie_details,
                      dbms_json.format_hierarchical,
                      dbms_json.pretty)
FROM   movies;
```

Automatically analyze and present JSON data as ”table”

- Schema structure translation to rows and columns
- Instantaneous relational access without any human intervention

Partial updates to JSON documents

```
UPDATE movies m
SET  m.data =  json_transform(m.data,
     set    '$.budgetUnit'= 'Million USD',
     set    '$.budget'= (m.data.budget / 1000000),
     remove '$.wiki_article',
     append '$?(@.views > 5000).awards' = 'popularMovie');
```

- Optimized in-place update of JSON documents
- Only update on disk the parts that have been changed
- For larger documents, can provide significant performance improvement over other document stores

Indexing

Single value index –
document indexed once
`$.firstName`

Multivalue index –
One document indexed multiple times
`$.phoneNumbers.number`

```
{
        "firstName":"John" ,
        "lastName":"Smith",
        "age":25,
        "address":{
                "street":"21 2nd Street","city":"New Y
                "state":"NY","postalCode":"10021",
                "isBusiness":false
        },
        "phoneNumbers":[
                {"type":"home", "number":"212 555-1234" },
                {"type":"mobile", "number":"646 555-4567" }
        ],
        "lastUpdated":"2019-05-13T13:03:35+0000"
}
```

Multi-Value Index – 23c

```
CREATE TABLE jsontab (a NUMBER, doc JSON);
INSERT INTO jsontab VALUES (1,
 '{"credit_score": [710, 720, 710, 730]}');
INSERT INTO jsontab VALUES (2,
 '{"credit_score": [750, 730, 750, 750]}');

CREATE MULTIVALUE INDEX jsontab_idx
    ON jsontab t
    (t.doc.credit_score.number());

SELECT COUNT(*)
FROM jsontab t
WHERE  JSON_EXISTS(t.doc,
                   '$.credit_score?(@ == 750)');
COUNT(*)
----------
 1
```

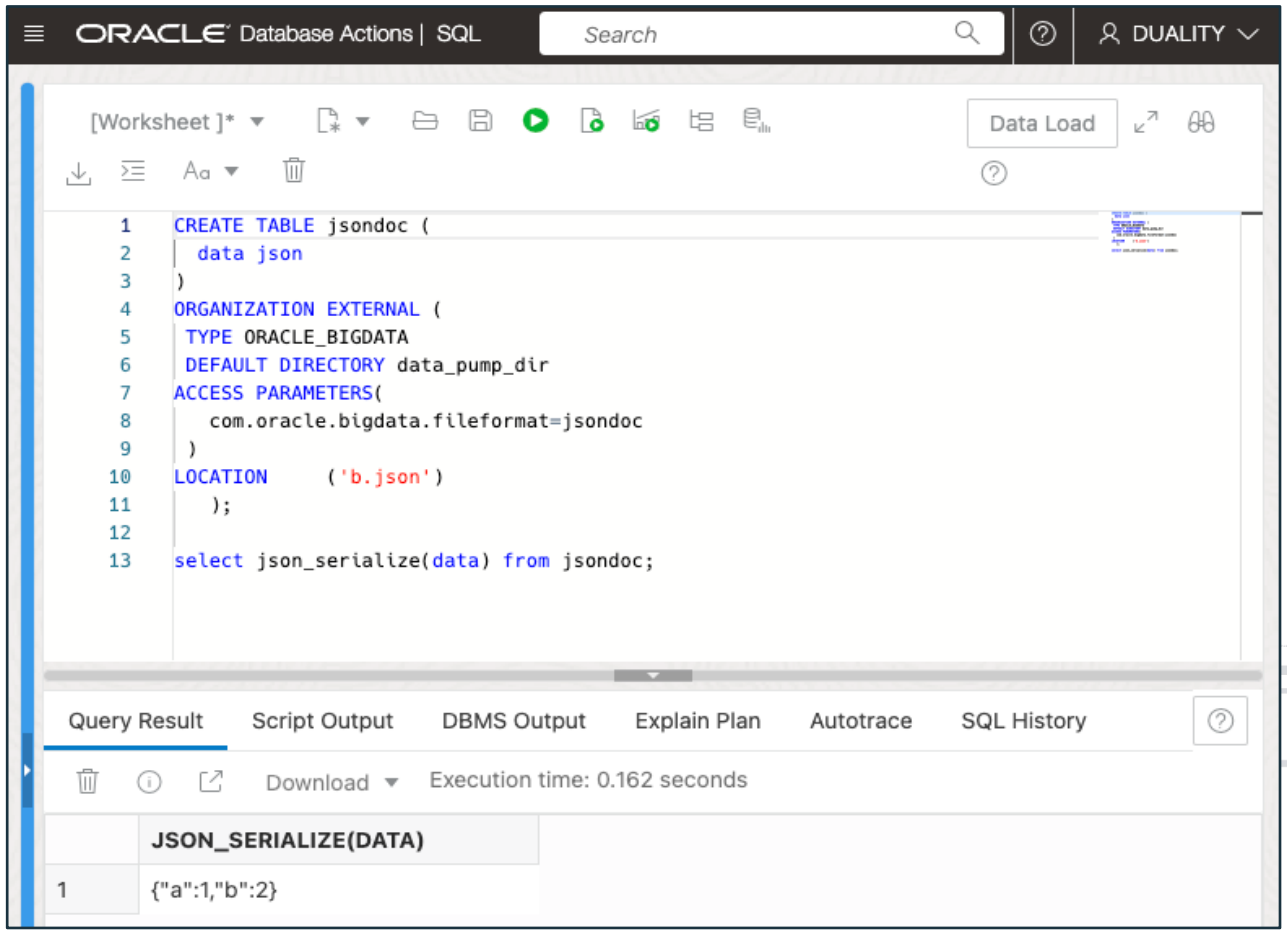
Multi-Value Index

- Allows to index JSON Array values
- Improved DML performance
- On-par query performance with current existence join plan with parent collection table
- Relies on B-Trees
- 19c workaround: JSON_Table materialized views

Support for JSON in External Tables

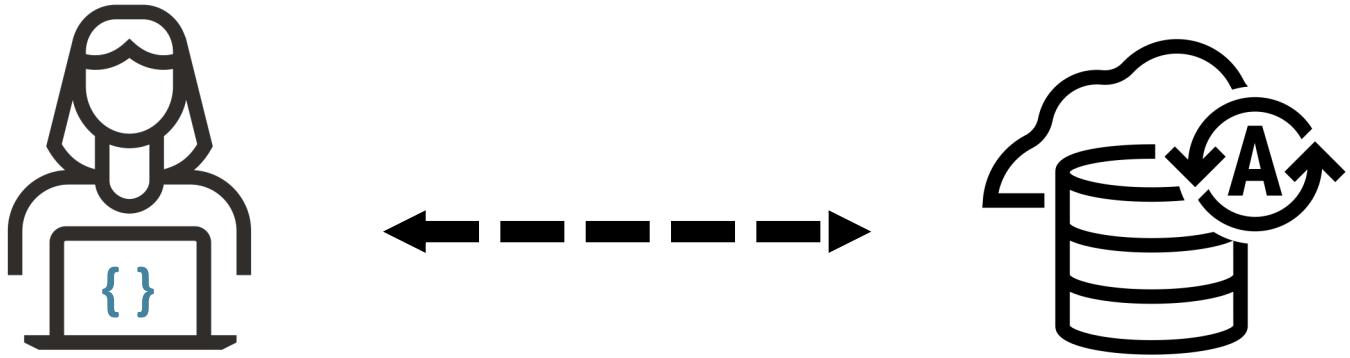
23c

- Access JSON files
  - Filesystem
  - Object Storage
- Supports multiple formats automatically
  - Line-delimited
  - Array-wrapped JSON
  - Nested JSON
- Exposed as JSON type column



JSON Collections

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

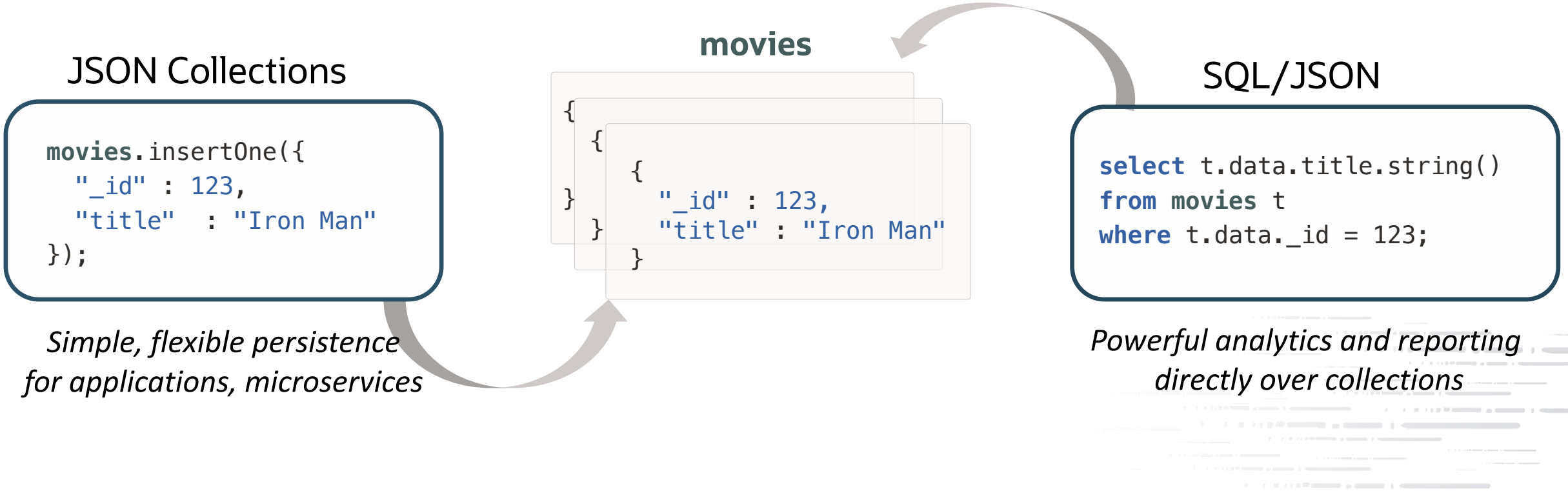
Filter expressions are executed as SQL over the backing table. Fully utilizes core Oracle Database features such as indexing, cost-based optimization, etc.

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

MongoDB Example

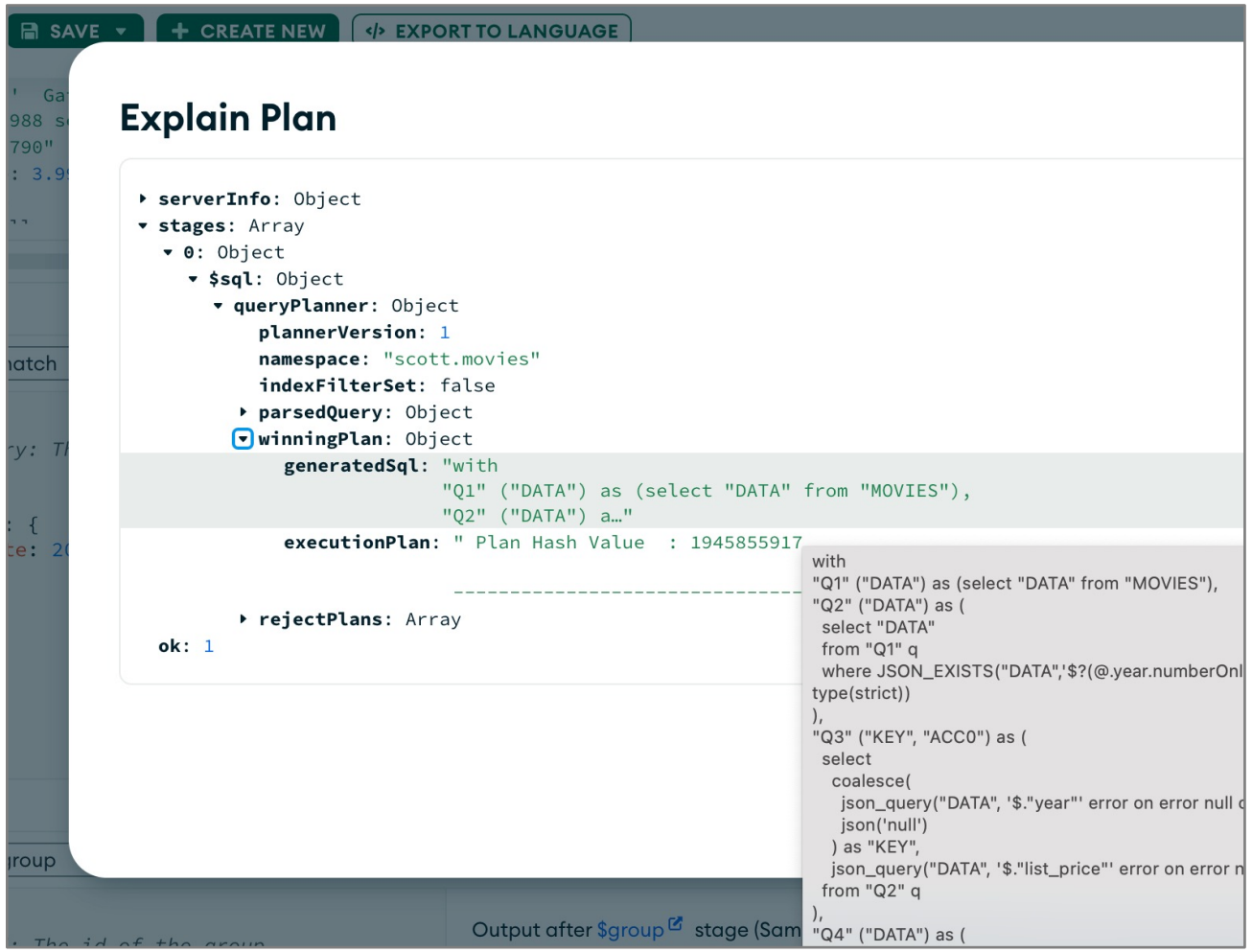
```
MongoClient mongoClient = MongoClients.create(connString);
MongoDatabase database = mongoClient.getDatabase("admin");

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…



New feature! MongoDB API support for **Aggregation Pipelines**

23c



Translated to Oracle analytical SQL

- Mongo pipelines operations equivalent to SQL operations
- Translation to SQL streamlines processing and improves performance

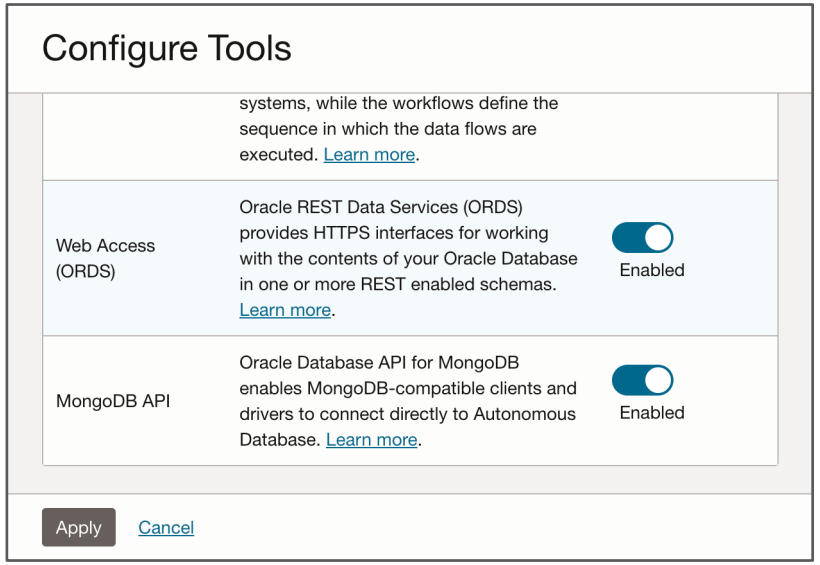
Supports commonly stages and expressions
(`$group, $match, $project, $sort,` etc.)

Available with **ORDS 23.2** for
Oracle Database 23c Free **later this month**

Using the Mongo API with Autonomous Database Serverless

① Enable network access

② Enable the feature

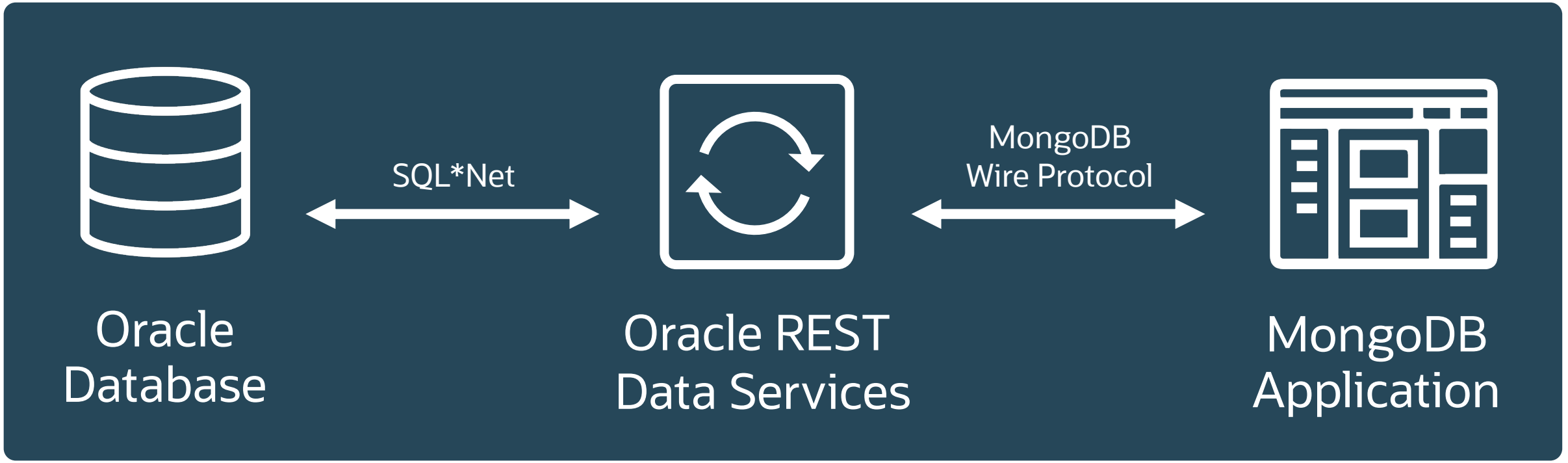


On by default for Autonomous JSON

③ Enable a user

```
create user myuser
  identified by "StrongPass123!";

grant dwrole to myuser;
grant unlimited tablespace
  to myuser;

exec ords.enable_schema(
    P_SCHEMA => 'MYUSER'
);
```

Using the Mongo API with Other Versions of Oracle Database



Autonomous Dedicated, Exadata Database Service, DBaaS, On-premise, etc.

MongoDB API limitations

- MongoAPI uses Oracle database authentication and authorization
  - *use Oracle database username/password only*

- Authenticate as the user corresponding to the database being accessed.
  - *e.g. authenticate as ADMIN, use ADMIN schema/database*

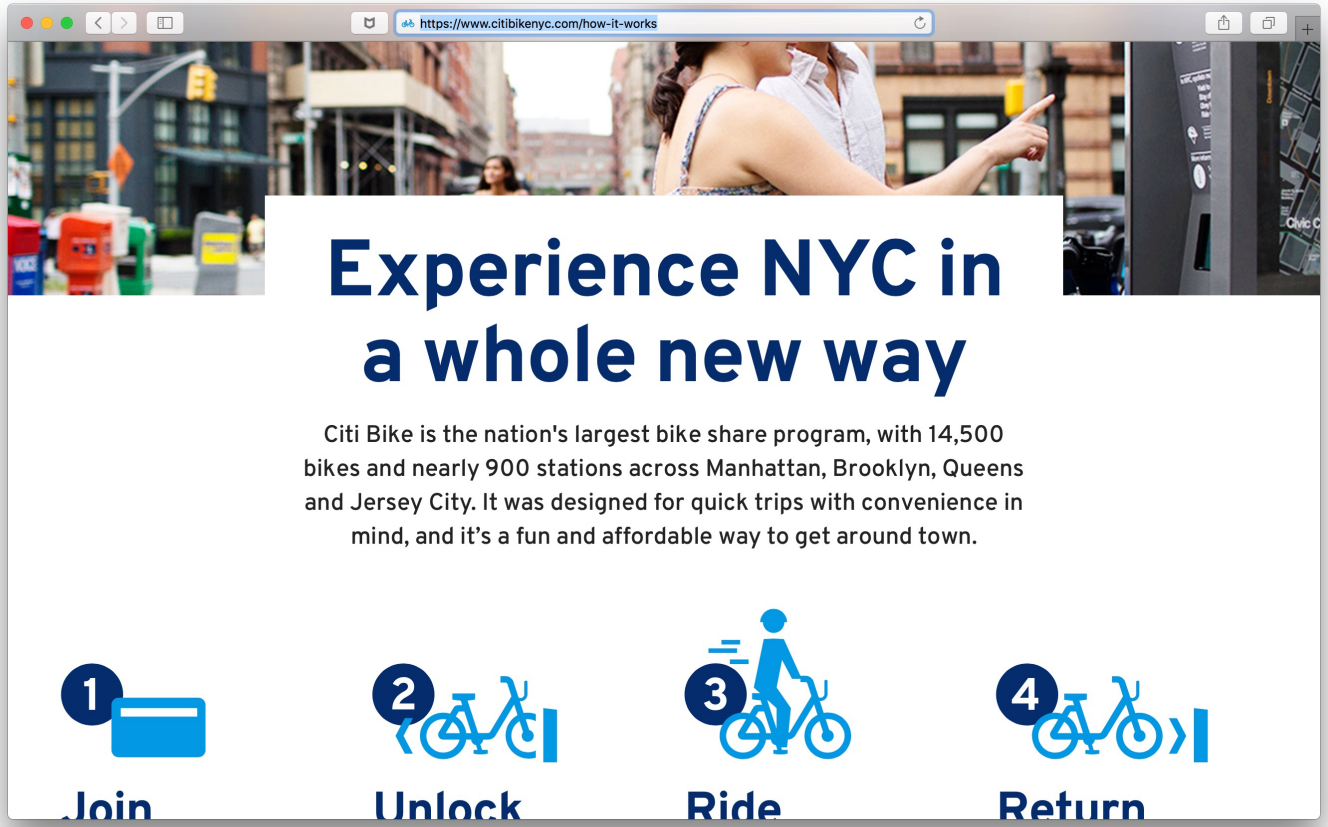
- Indexes must be created from SQL (in release 19c, will be supported in 23c)
  - *db.createIndex() from MongoAPI does not create an index*

```
SQL> create index lastidx on
       employee (json_value(data, '$.job'  error on error));
```

- Aggregation pipeline is only supported on 23c

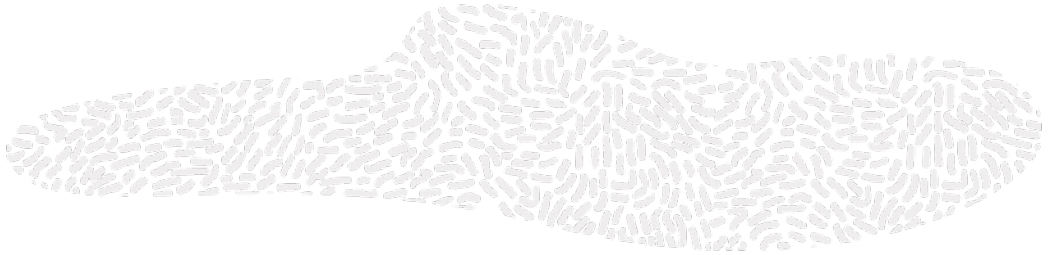
Demonstration

Citi Bike



SOURCE:
https://www.citibikenyc.com/

Bike Station Data



station_information.dat


```
{
    "capacity": 55,
    "eightd_has_key_dispenser": false,
    "eightd_station_services": [],
    "electric_bike_surcharge_waiver": false,
    "external_id": "66db237e-0aca-11e7-82f6-3863bb44ef7c",
    "has_kiosk": true,
    "lat": 40.76727216,
    "lon": -73.99392888,
    "name": "W 52 St & 11 Ave",
    "region_id": "71",
    "rental_methods": [
        "CREDITCARD",
        "KEY"
    ],
    "rental_url": "http://app.citibikenyc.com/S6Lr/I....",
    "short_name": "6926.01",
    "station_id": "72",
    "station_type": "classic"
}
```

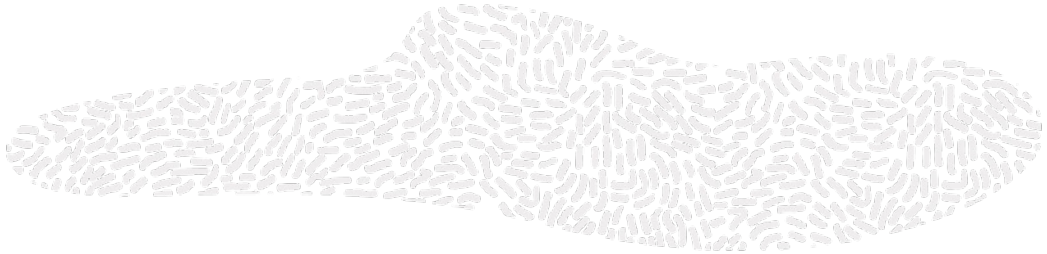

1

N

station_status.dat


```
{
    "eightd_has_available_keys": false,
    "is_installed": 1,
    "is_renting": 1,
    "is_returning": 1,
    "last_reported": 1563207770,
    "last_updated": 1563207810,
    "num_bikes_available": 1,
    "num_bikes_disabled": 0,
    "num_docks_available": 52,
    "num_docks_disabled": 2,
    "num_ebikes_available": 0,
    "station_id": "72"
}
```

Oracle's Strategy on JSON



Leader: first enterprise relational database with JSON support: 12.1.0.2 (2014)

- large number of customers, NoSQL migrations, 26% of Autonomous DB use JSON

Open: JSON enhancement are contributed to the open standards

- SQL standard: SAP Hana, IBM DB2, Postgres, MySQL,… follow Oracle
- JSON Schema: Oracle wrote the database vocabulary

Innovative: many new industry leading JSON features

- JSON_Dataguide to derive JSON schema from data, detect schema changes, auto create relational views
- JSON Schema to validate JSON but also describe database objects, constraints
- JSON Relational Duality, GraphQL, …

Thank you