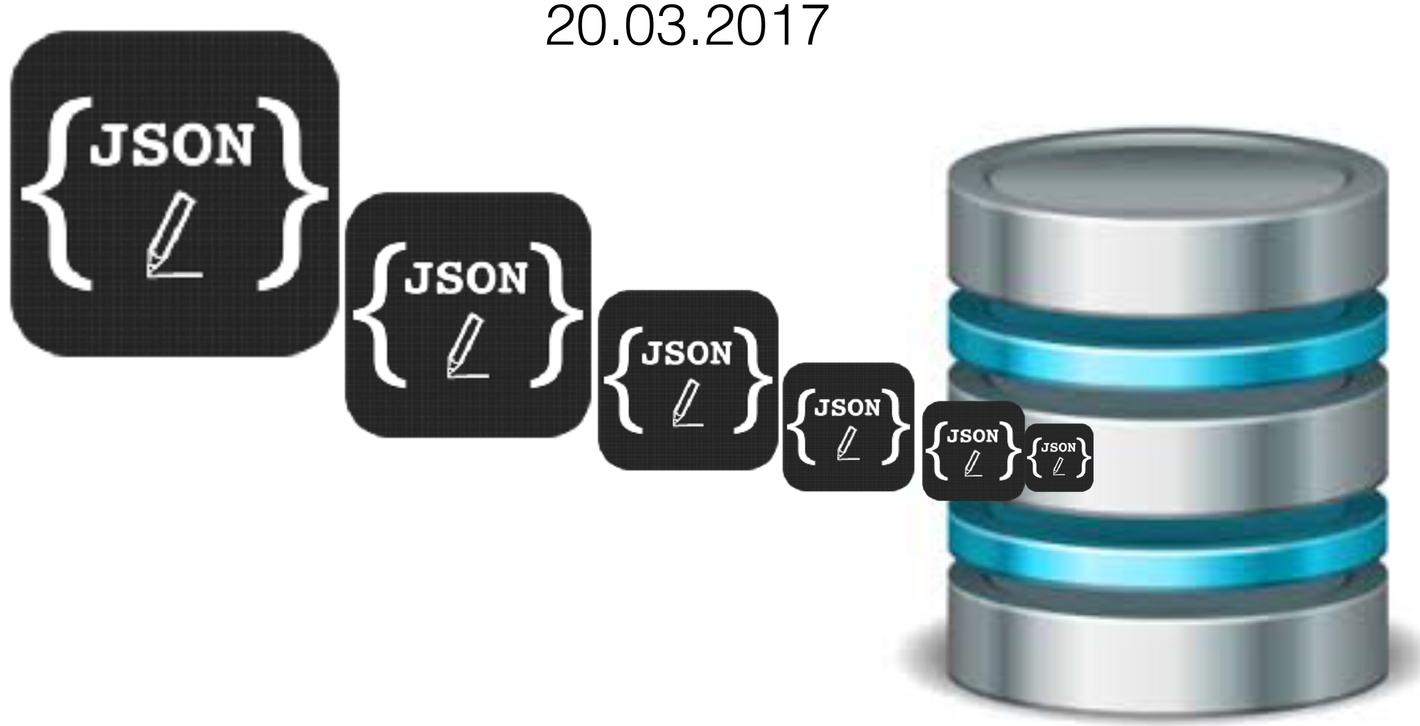
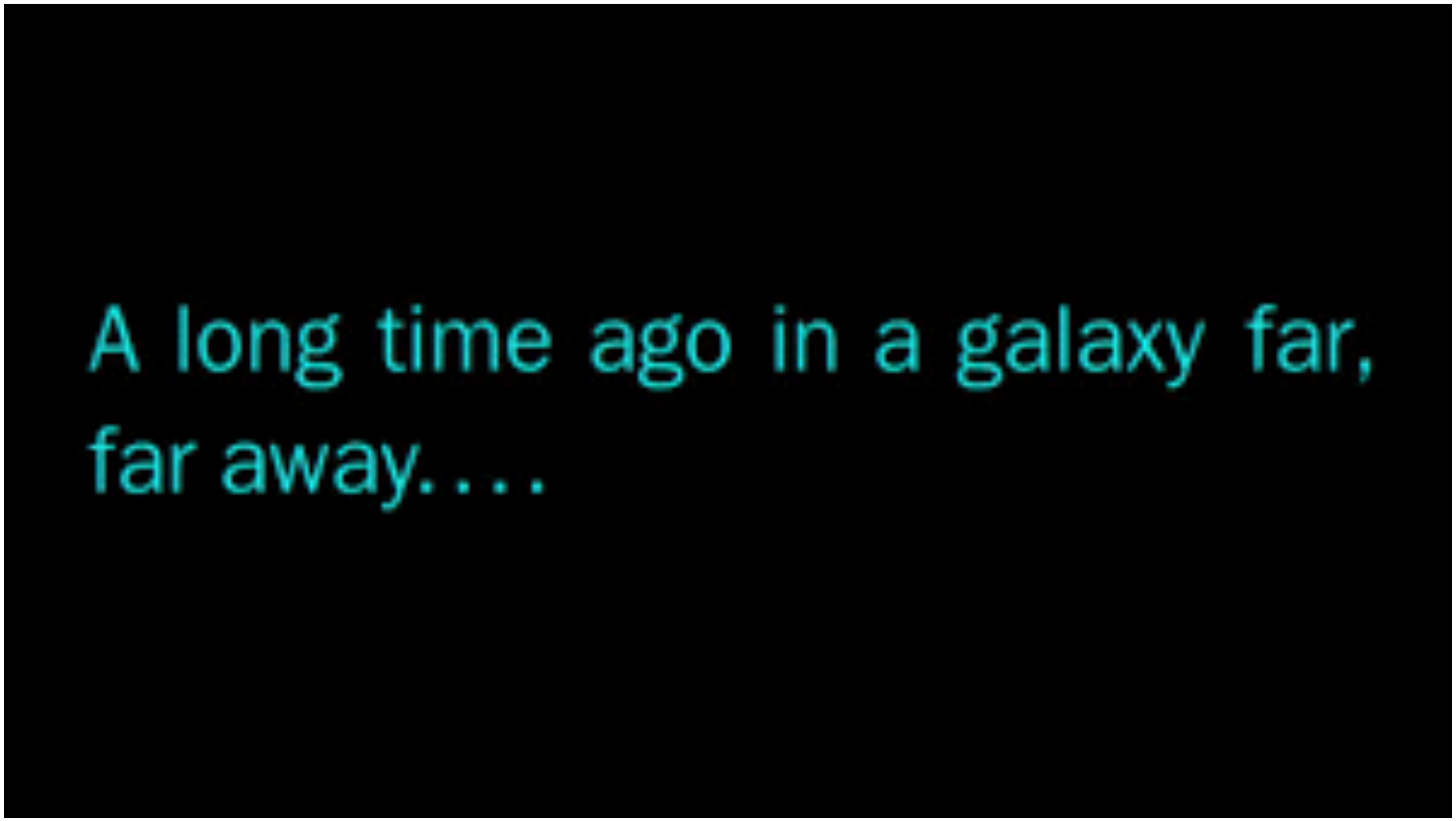
JSON Schema-less into RDBMS

Most of the material was taken from the Internet and the paper “JSON data management: sup- porting schema-less development in RDBMS”, Liu, Z.H., B. Hammerschmidt, and D. McMahon, 2014

Agustín Zúñiga.

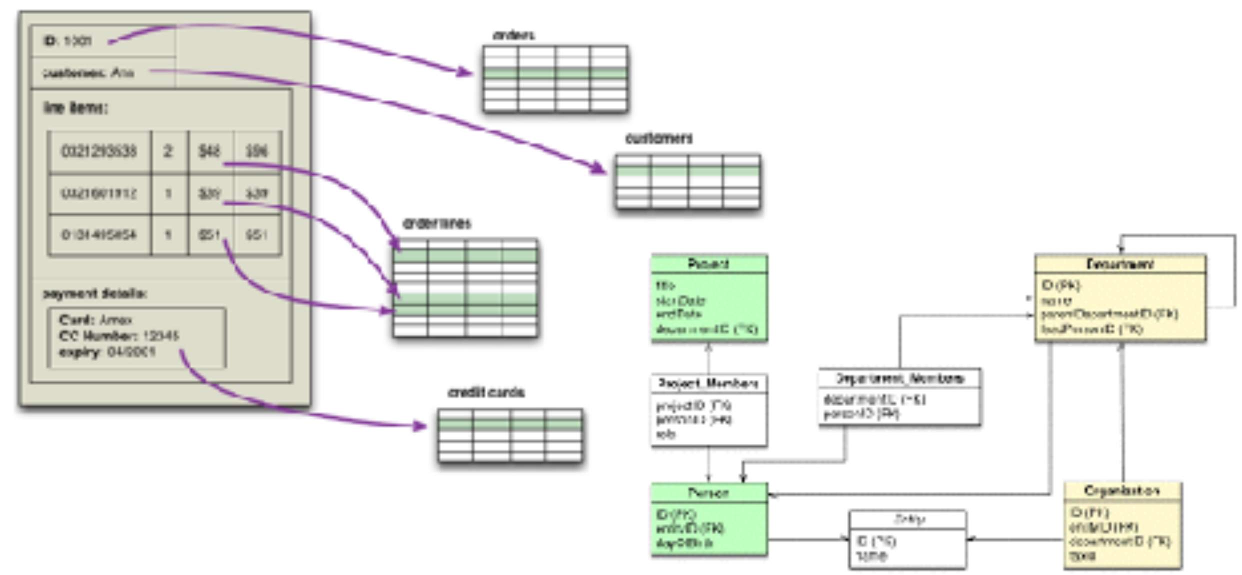
20.03.2017




A long time ago in a galaxy far,
far away....

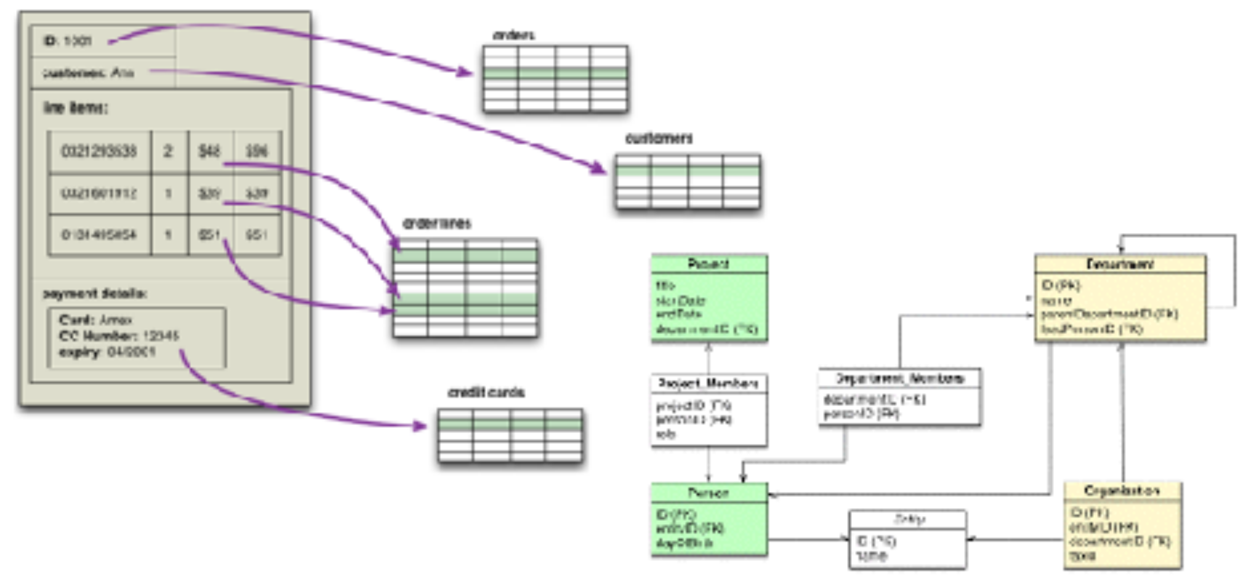
Classic data management

- Data collections.
- Relational Data Base Systems **RDBMS**.
- **Well-structured** data interaction.



Classic data management

- Data collections.
- Relational Data Base Systems **RDBMS**.
- **Well-structured** data interaction.
- **However since the beginning of the XX Century…**

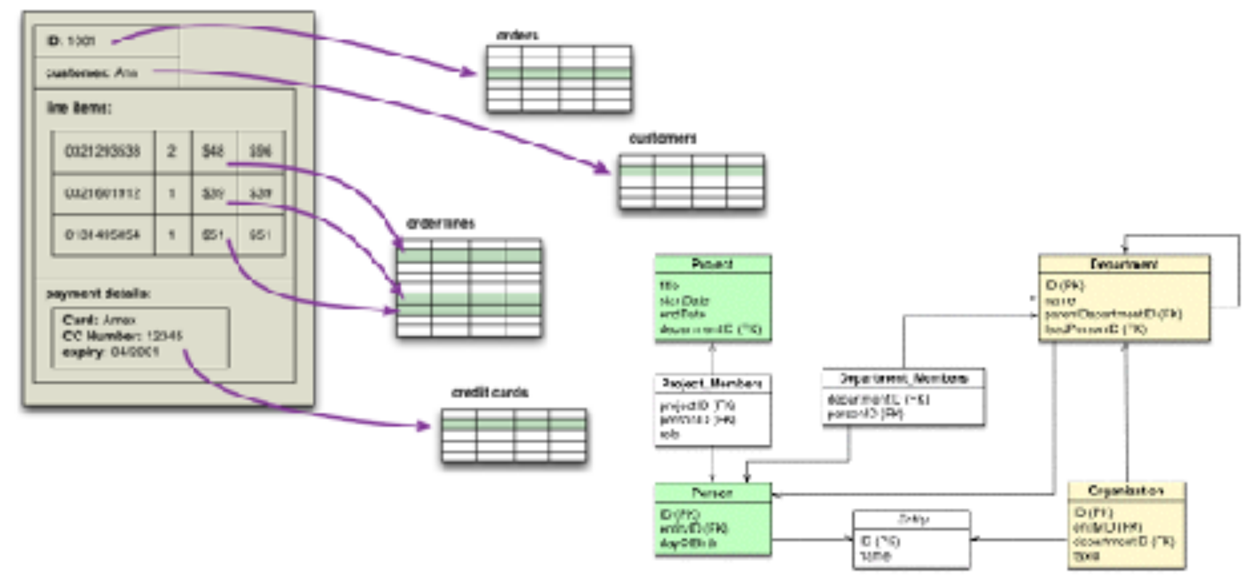


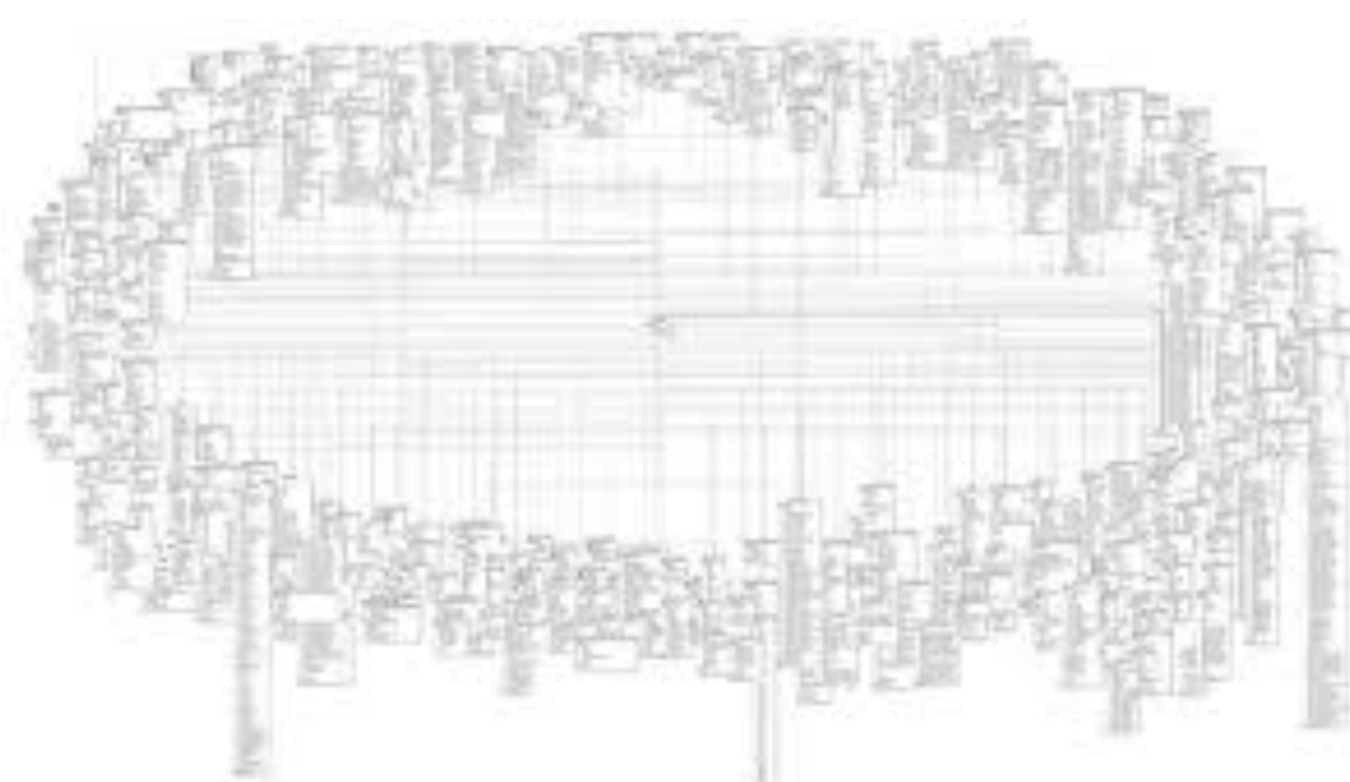
Classic data management

- Data collec
- Relational I
  Systems **R**
- **Well-struc**
  interaction.
- **However s**
  beginning
  Century…

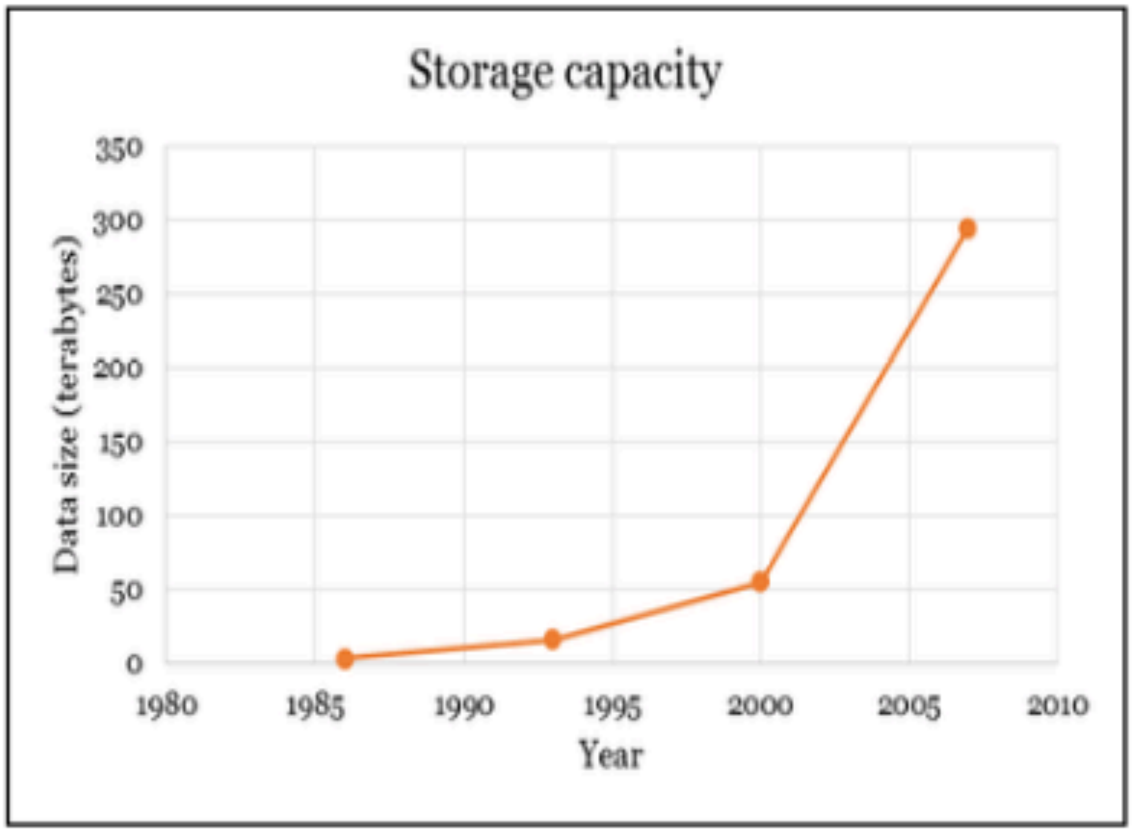
Classic data management

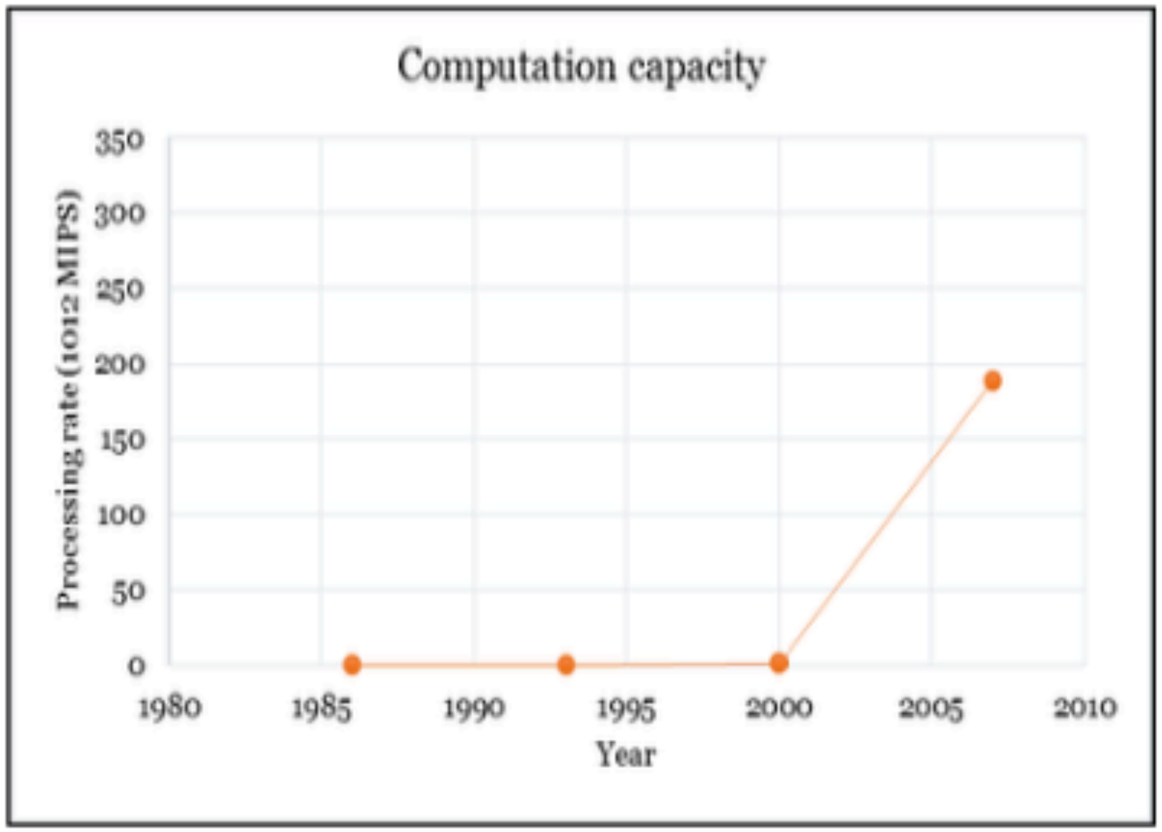
- Data collections.
- Relational Data Base Systems **RDBMS**.
- **Well-structured** data interaction.
- **However since the beginning of the XX Century…**





Storage capacity vs. Computation Capacity

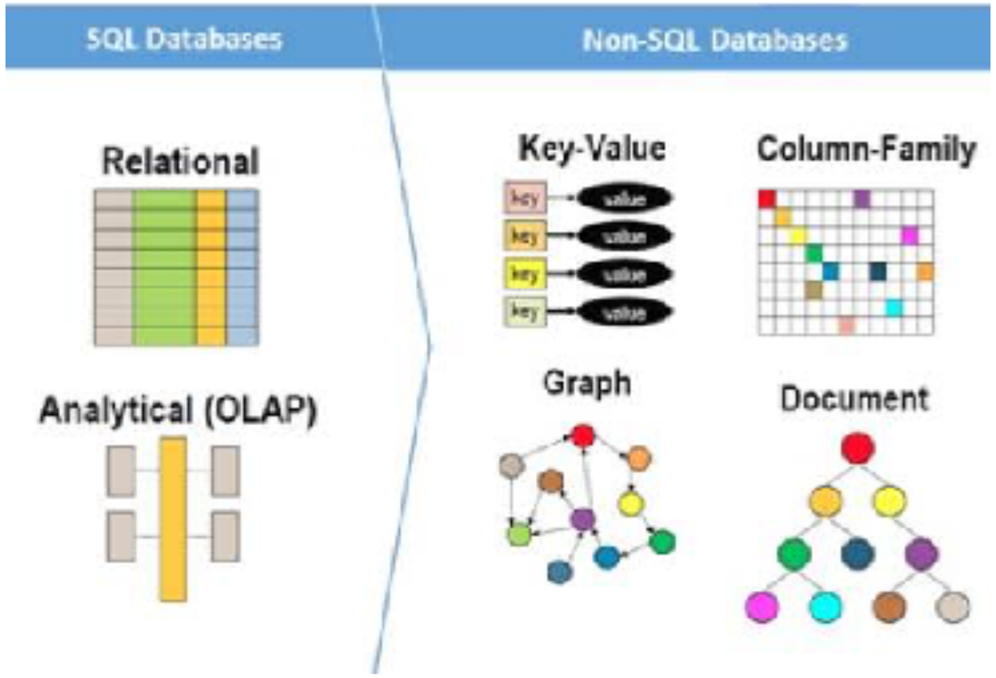




Sharma, S., An Extended Classification and Comparison of NoSQL Big Data Mod- els. arXiv preprint arXiv:1509.08035, 2015.

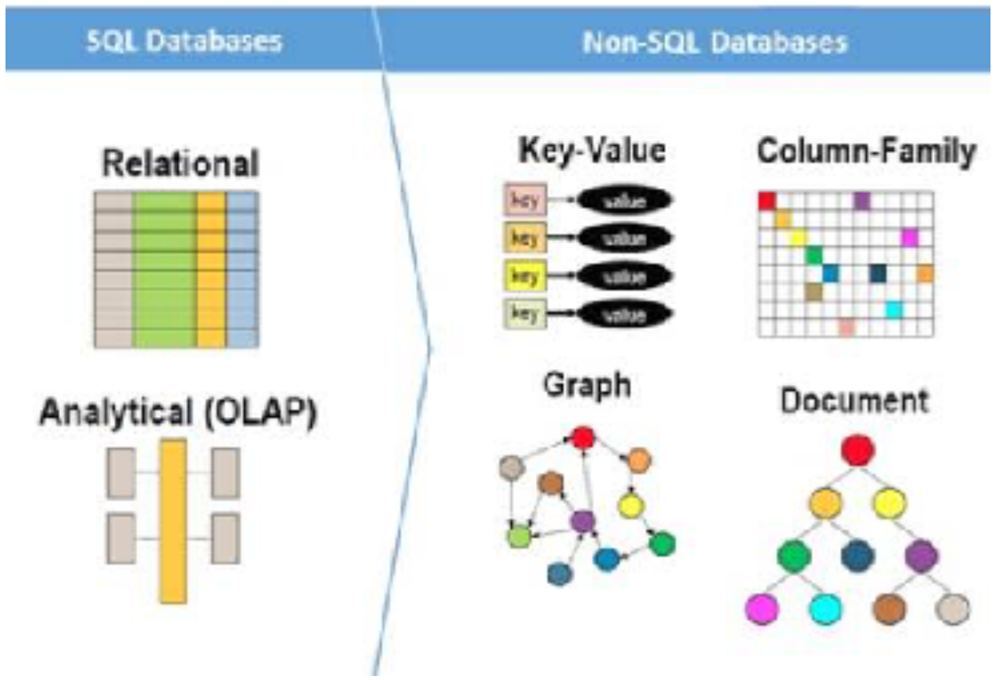
Data management…a new approach

- More and more data collections.
- **Well-structured** data + **Semi-structured** data + **unstructured** data.
- Of course Relational Data Base Systems **RDBMS**

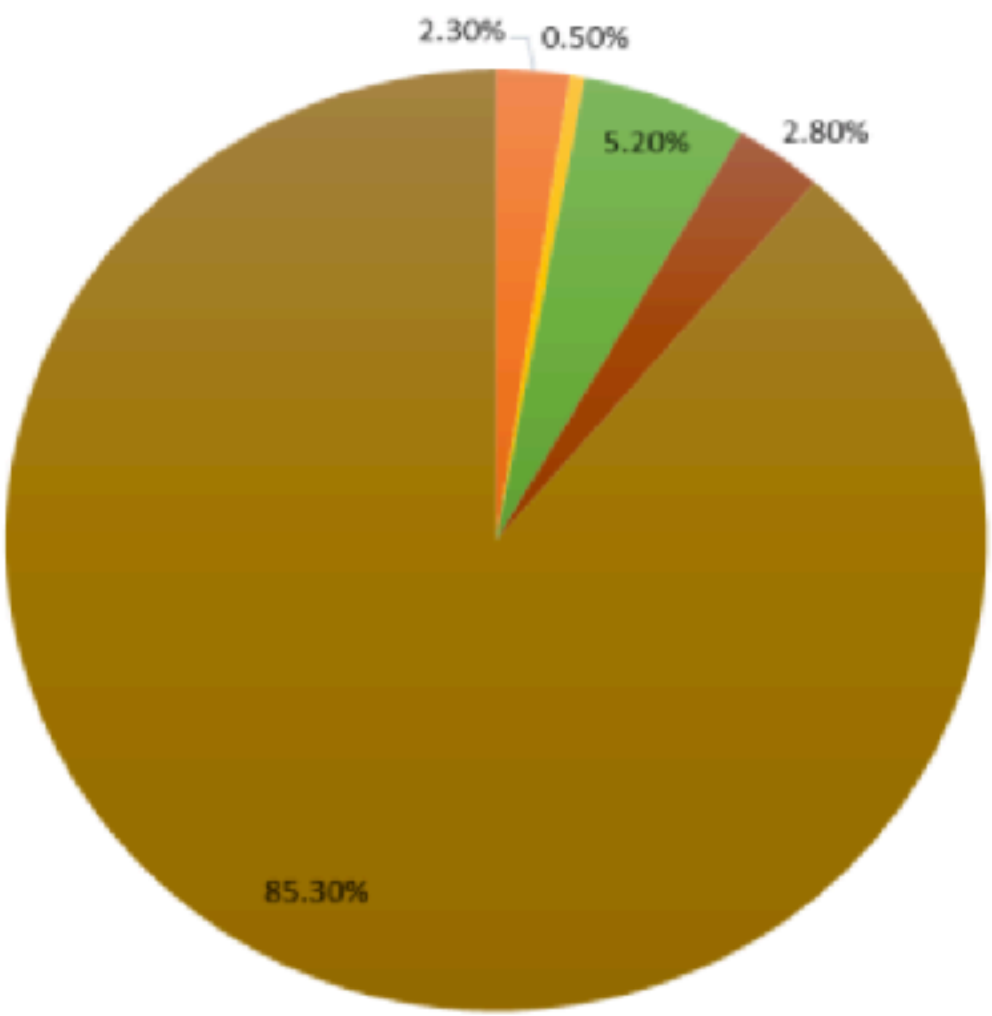


Data management…a new approach

- More and more data collections.
- **Well-structured** data + **Semi-structured** data + **unstructured** data.
- Of course Relational Data Base Systems **RDBMS,** but also **NoSQL tools.**



Ranking of various types of data models

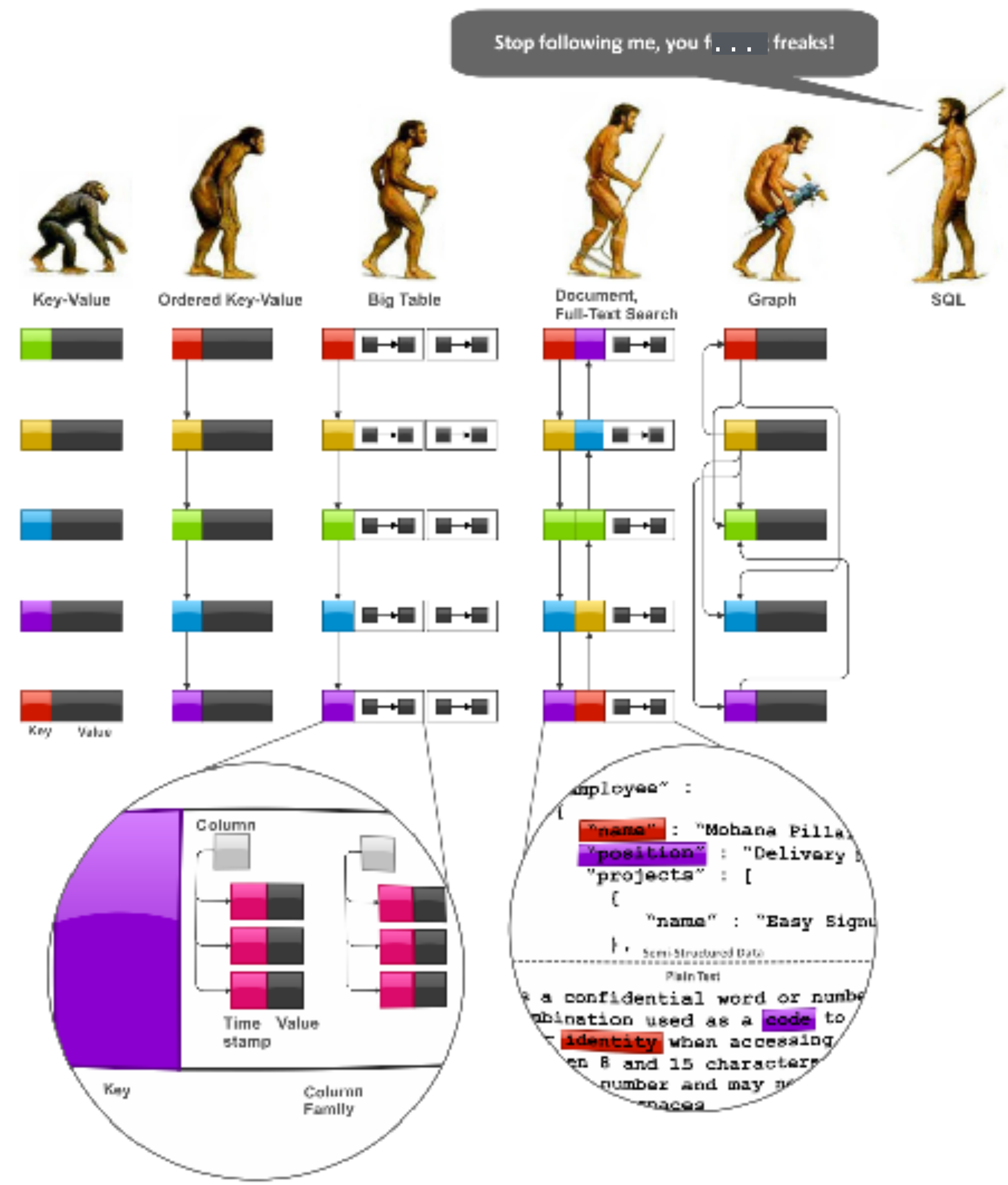


85.30%	Relational DBMS	SQL
5.20%	Document stores	NoSQL
2.80%	Key-value stores	NoSQL
2.30%	Wide column stores	NoSQL
0.50%	Graph DBMS	NoSQL

Sharma, S., An Extended Classification and Comparison of NoSQL Big Data Mod- els. arXiv preprint arXiv:1509.08035, 2015.

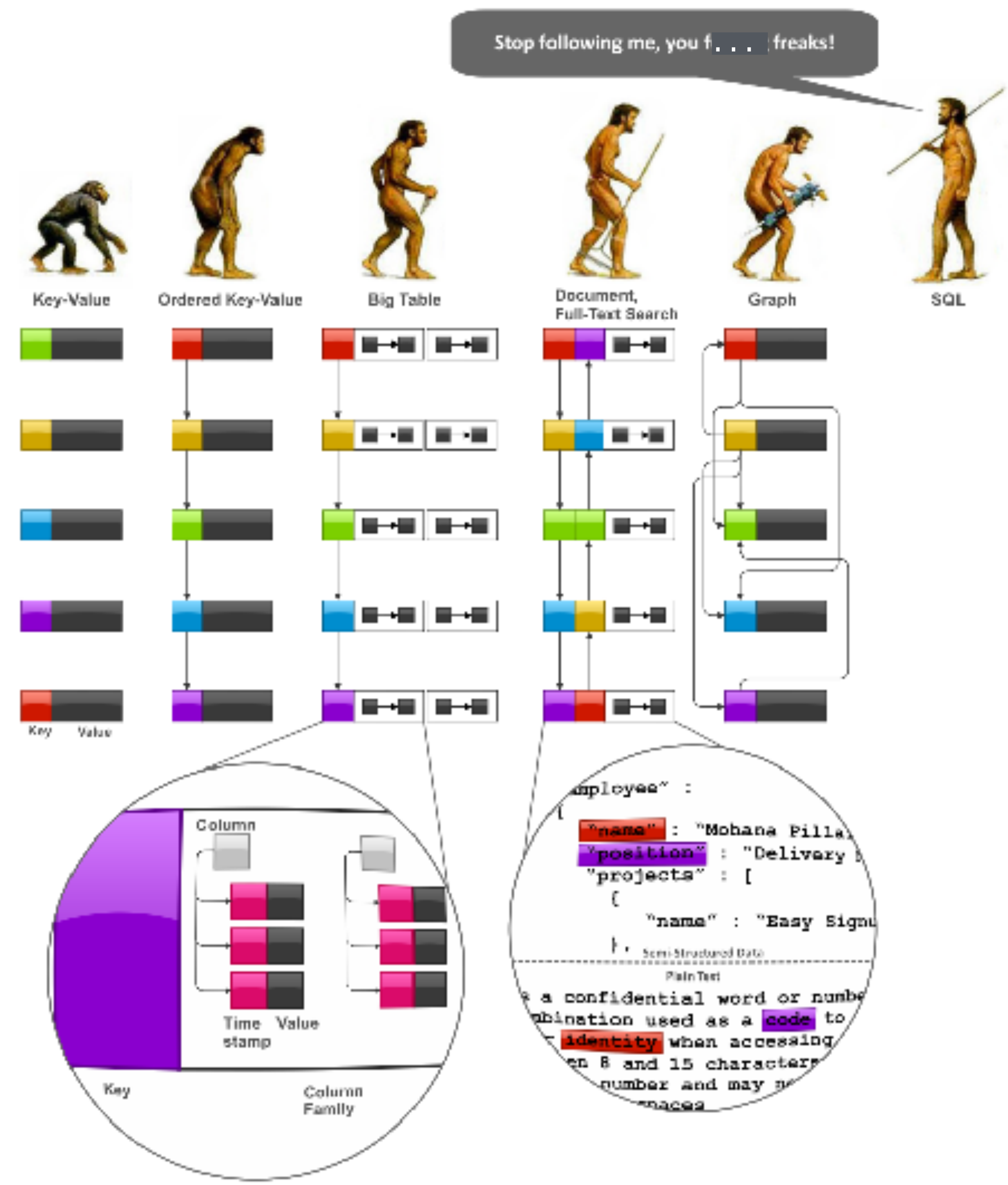
Schema-less in RDBMS

- Why?
- Goal: **Integrated data management.**
- Specific target: document stores, **JSON data.**



Schema-less in RDBMS

- Why?
- Goal: **Integrated data management.**
- Specific target: document stores, **JSON data.**
- Some previous ideas:
  - Shredding objects.
  - *New SQL data types.*
  - *Indexing performance.*
- Very demanding for developers and…

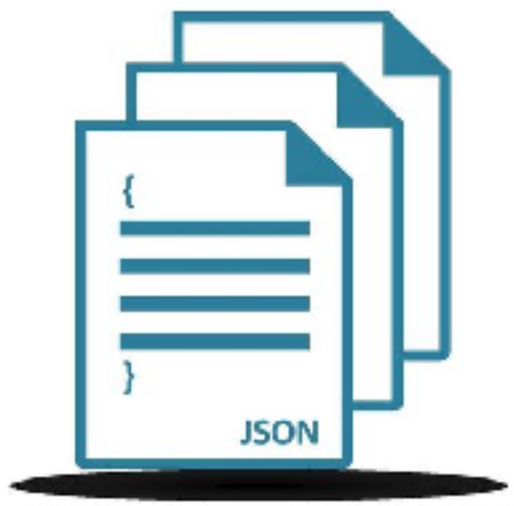


How to manage JSON Schema-less in RDBMS?

EID	Fname	Lname	AddID
1	Saeed	Rahimi	15
2	Bhabani	Misra	12
3	Bard	Rubin	13
4	Frank	Haug	11
6	Chih	Li	13

Relational data model

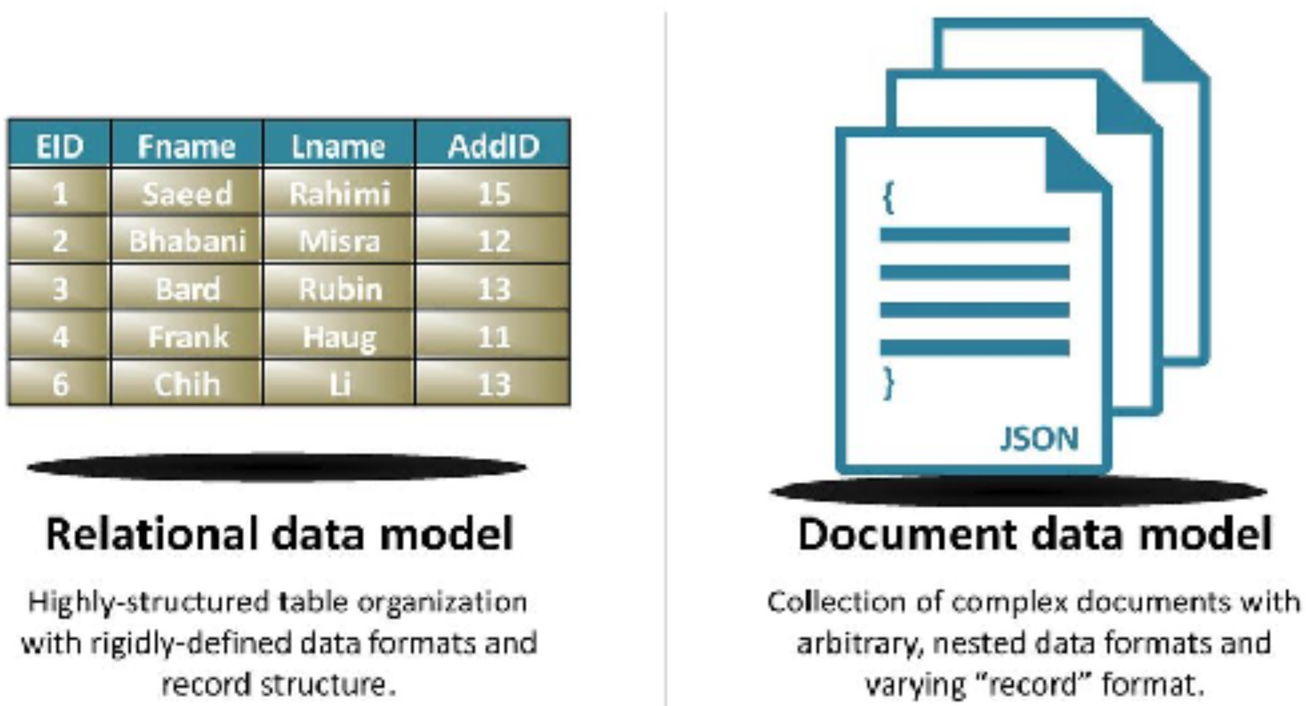
Highly-structured table organization with rigidly-defined data formats and record structure.



Document data model

Collection of complex documents with arbitrary, nested data formats and varying "record" format.

How to manage JSON Schema-less in RDBMS?



3 Principles for JSON:

1. **Storage**
2. **Query**
3. **Index**

Why 3 principles?

Requirements of JSON data management in RDBMS:

- **Data Modeling Difference:** Schema First Versus Schema Later/Never.
- **Querying Difference:** Querying Flattened Data with Static Schema Versus Querying Hierarchical Object with Dynamic Schema.
- **Indexing Difference:** Partial-Schema-aware Indexing Method Versus Schema agnostic Indexing Method

1st Principle:
Storage Principle for JSON

Data Modeling Difference

Schema First

- RDBMS.
- Structures (**SCHEMA**) can be cleanly separated from the content.
- Entity/Relationship (E/R) modelling: primary and foreign **keys** to support join pattern.
- Structural **changes** require DDL statements to alter the system meta-data before new data with the changed shape can be loaded.

Data Modeling Difference

Schema First

- RDBMS.
- Structures (**SCHEMA**) can be cleanly separated from the content.
- Entity/Relationship (E/R) modelling: primary and foreign **keys** to support join pattern.
- Structural **changes** require DDL statements to alter the system meta-data before new data with the changed shape can be loaded.

Schema Later/Never.

- Structure not easily separable: varies from instance to instance. Some issues:
- **Sparse-attribute:** collection of objects may have large number of sparsely populated columns when stored relationally. NULL
- **Polymorphic typing**: datatype instances
- **Singleton-to-collection**: cardinality vary from one instance to another, or over time.
- **Recursive structure:** not first class operations in SQL.

Support for Document-Object Store Model without Relational Shredding (1)

No shredding. No relational decomposition.

- A JSON object collection is modeled as a **table** having **one column** storing **JSON objects**. **Each row** in the table holds a **JSON object instance** in the collection.

JSON_[1 to n]
JSON_instance_1
JSON_instance_2
…
JSON_instance_n

Support for Document-Object Store Model without Relational Shredding (1)

No shredding. No relational decomposition.

- A JSON object collection is modeled as a **table** having **one column** storing **JSON objects**. **Each row** in the table holds a **JSON object instance** in the collection.

JSON_[1 to n]
JSON_instance_1
JSON_instance_2
…
JSON_instance_n

No JSON Sql datatype. No changes in RDBMS kernel neither API's clients…

- JSON data storage mapped as external "table" to RDBMS. **Consume it as is.**

Support for Document-Object Store Model without Relational Shredding (2)

Storing JSON using existing SQL datatypes with Check Constraint

- **JSON data** in SQL VARCHAR, CLOB, RAW, or BLOB columns, instead of abstract SQL datatype.
- IS JSON: SQL built-in operator to **verify** if input text or binary is a valid JSON object before storing.
- Ensures full operational completeness for JSON data: partitioning, replication, import/export, and bi-temporal features.

Support for Document-Object Store Model without Relational Shredding (2)

Storing JSON using existing SQL datatypes with Check Constraint

- **JSON data** in SQL VARCHAR, CLOB, RAW, or BLOB columns, instead of abstract SQL datatype.
- IS JSON: SQL built-in operator to **verify** if input text or binary is a valid JSON object before storing.
- Ensures full operational completeness for JSON data: partitioning, replication, import/export, and bi-temporal features.

Virtual Relational Columns over JSON object collection.

- For partial schema: **data** in a JSON object collection can be **projected out** as virtual columns attaching to the collection table using the JSON_VALUE().

Support for Document-Object Store Model without Relational Shredding (2)

T1	*CREATE TABLE shoppingCart_tab (shoppingCart VARCHAR2(4000) check (shoppingCart is JSON), sessionId NUMBER AS (JSON_VALUE(shoppingCar ,'$.sessionId' RETURNING NUMBER)) VIRTUAL, userlogin varchar2(30) AS (CAST(JSON_VALUE(shoppingCart,,'$.userLoginId') AS varchar2(30))) VIRTUAL)*
INS1	*INSERT INTO shoppingCart_tab(shoppingCart) VALUES(' {"sessionId" : 12345, "creationTime": "12-JAN-09 05.23.30.600000 AM", "userLoginId" : "johnSmith3@yahoo.com", "Items" : [{"name": "iPhone5", "price" :99.98, "quantity" :2, "used": true, "comment" : "minor screen damage"}, {"name": "refrigerator", "price" :359.27, "quantity" :1, "weight": 210, "Height" : 4.5, "Length" : 3, "manufacter": "Kenmore", "color" : "Gray"}]} ')*

2nd Principle:
Query Principle for JSON


FOX

Querying Difference

Flattened Data with Static Schema

- Data structures are decomposed relationally.
- First registering object structural schema as meta-data.
- Consequently, the structure of the data is known ahead of query execution time so that **only data is examined** during query execution time.

Hierarchical Object with Dynamic Schema

- Schema is not registered as system meta-data.
- Query language **needs to have constructs** that can query both the structure and data together.
- SQL alone is not sufficient

Querying Difference

Flattened Data with Static Schema

- E-R models hierarchies of data using the master-detail pattern: requires the use of **explicit join:**
  - Register object structural schema as meta-data.
  - The structure of the data is known ahead of query execution time so that **only data is examined** during query execution time.
- **Results** in columns values, always scalar values.

Hierarchical Object with Dynamic Schema

- For JSON objects which are not stored in decomposed manner, it is more natural to express hierarchical traversal using a **path navigation language**.
- SQL needs to embrace path navigation constructs for querying objects natively.
- It needs to support wildcard steps in the path and traversal of hierarchical structures.
- **Results** represent projections of both scalars and sub-structures from underlying JSON objects.

Leverage SQL as inter-JSON object Set-oriented Query Language

- Instead of 'Structured Query Language', we call SQL as 'Set oriented Query Language'.
- SQL can be used as an inter-object query language to query objects stored in the object collection table.
- No theoretical reason to construct yet another set-at-a- time query language.
- JSON path language needed to provide declarative navigational access of JSON object instance in conjunction with SQL.

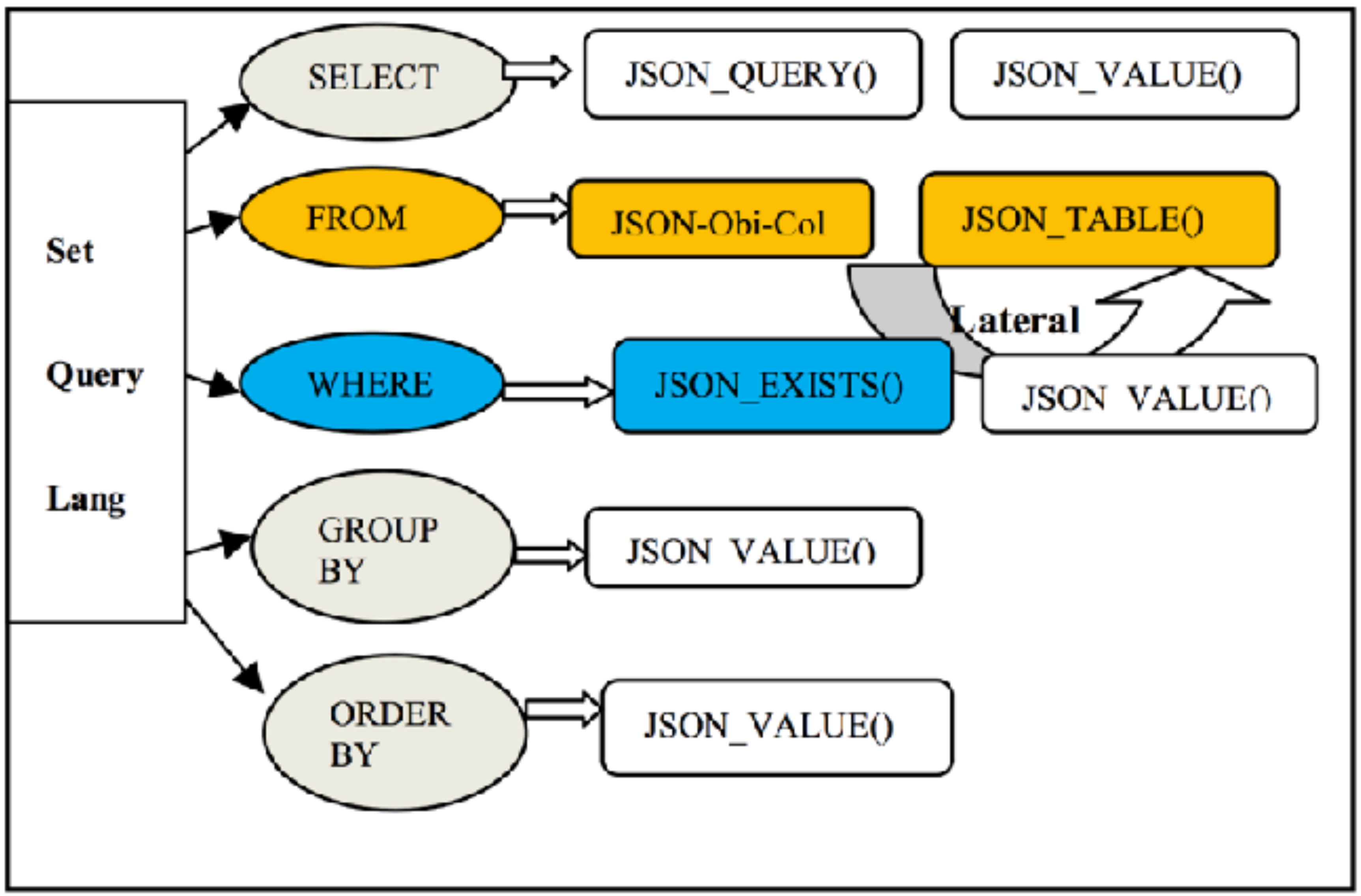
SQL/JSON Standard

- Defines a set of SQL/JSON query operators that embed a simple JSON path language to provide declarative query language over a collection of JSON objects and a set of SQL/JSON construction functions from pure relational data.

SQL/JSON Standard

- Defines a set of SQL/JSON query operators that embed a simple JSON path language to provide declarative query language over a collection of JSON objects and a set of SQL/JSON construction functions from pure relational data.

SQL/JSON operators usage in SQL



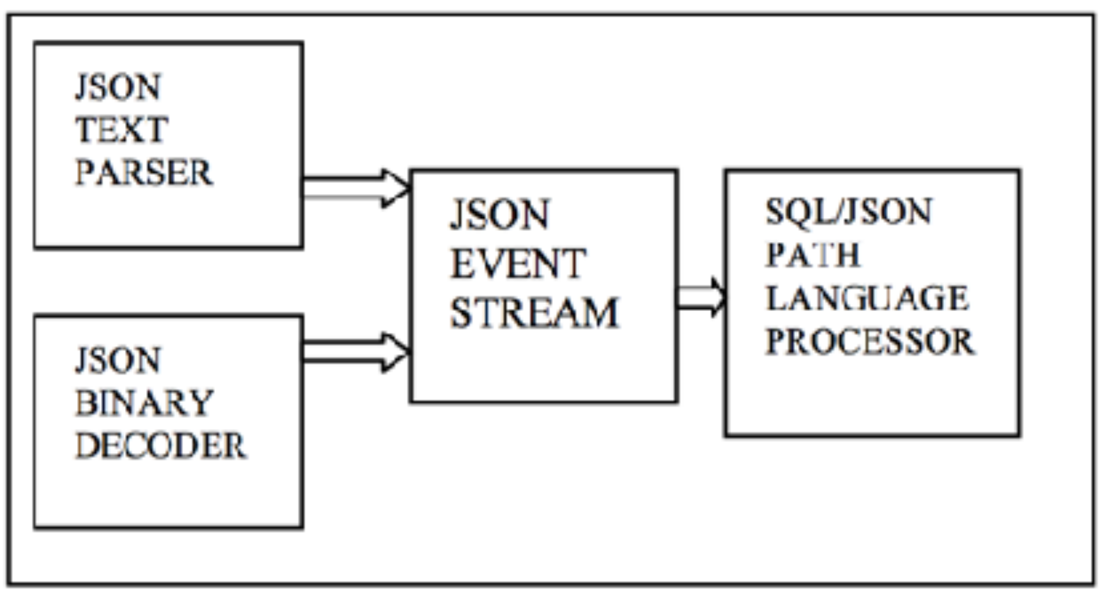
SQL/JSON Path Language

- Path step expressions with filter expressions used as predicates for path steps.
- Each path step expression is either the JSON object member accessor or the JSON array element accessor.
- Sequence Data / Model Predicate Filter expression.
- Lax Mode for object and array accessor / Lax Error Handling.

Q1	*SELECT p.sessionId, **JSON_QUERY**(p.shoppingCar, '$.items[1]' RETURN AS VARCHAR(2000))* *FROM shoppingCart_tab p* *WHERE **JSON_EXISTS**(p.shoppingCar , '$.item?(name="iPhone"))')* *ORDER BY p.userlogin*
Q2	*SELECT p.sessionId, p.userlogin, v.Name, v.price, v.Quantity* *FROM shoppingCart_tab p,* ***JSON_TABLE**(p.shoppingCart , '$.items[*]'* *COLUMNS* *Name VARCHAR(20) PATH '$.name',* *price number PATH '$.price',* *Quantity integer PATH '$.Quantity') v*

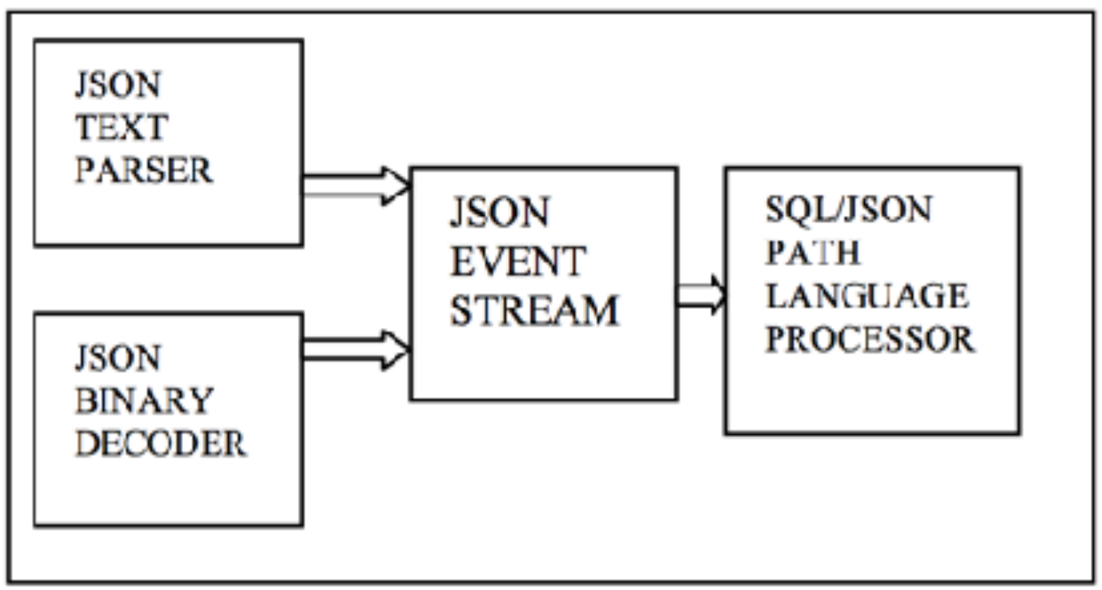
SQL/JSON in Oracle

- Support **streaming processing** without materializing entire JSON objects in memory



SQL/JSON in Oracle

- Support **streaming processing** without materializing entire JSON objects in memory



Qry#	Original Query	Transformed Query for optimization
T1	*SELECT p.sessionId, p.creationTime, v.Name, v.price, v.Quantity FROM shoppingCart_tab p, **JSON_TABLE**(p.shoppingCart , '$.items[*]' COLUMNS Name VARCHAR(20) PATH '$.name', price number PATH '$.price', Quantity integer PATH '$.Quantity') v*	*SELECT p.sessionId, p.creationTime, v.Name, v.price, v.Quantity FROM shoppingCart_tab p, **JSON_TABLE**(p.shoppingCart , '$.items[*]' COLUMNS Name VARCHAR(20) PATH '$.name', price number PATH '$.price', Quantity integer PATH '$.Quantity') v **WHERE JSON_EXISTS(p.shoppingCart, '$.items[*]')***

3rd Principle:
Index Principle for JSON

Index Difference

Partial-Schema-aware

- Used to define virtual columns and relational views on top of the collection.
- Virtual columns can be used to construct indexes and boost the performance of queries of **known patterns**.
- Natural consequence of adopting the **'data first, schema later'**
- **RDBMS: 'schema first, index definition later'.**

Schema agnostic

- JSON object in a row may have array consisting of multiple values to be indexed.
- Common indexing methods assume one indexed value per row: Not sufficient to index multiple values within an array of a JSON object: **index cardinality issue.**
- Indexing method should not rely on knowing any partial schema for the target collection: **JSON member + array names** to be indexed together with the data.
- Natural consequence of adopting the **'data first, schema never.'**

Partial-Schema-aware (1)

- Common path expressions or members can be identified for a collection.
- Partial schemas can be **projected out as auxiliary structures** on top of the original JSON object collection in the form of functional indexes, virtual columns.
- JSON_VALUE() : the **simplest functional indexing** method. Facilitate range searches on the result of JSON_VALUE()

Partial-Schema-aware (1)

- Common path expressions or members can be identified for a collection.
- Partial schemas can be **projected out as auxiliary structures** on top of the original JSON object collection in the form of functional indexes, virtual columns.
- JSON_VALUE() : the **simplest functional indexing** method. Facilitate range searches on the result of JSON_VALUE()
- If **multiple members** from a JSON object collection need to be range queried together, we can create a **composite** B+ tree **index** over multiple virtual columns, each of which is a projection of a member.

IDX	*CREATE INDEX shoppingCart_Idx ON shoppingCart_tab(userlogin, sessionId)*

Partial-Schema-aware (2)

Q2	*SELECT p.sessionId, p.userlogin, v.Name, v.price, v.Quantity* *FROM shoppingCart_tab p,* ***JSON_TABLE***(*p.shoppingCart , '$.items[*]'* *COLUMNS* *Name VARCHAR(20) PATH '$.name',* *price number PATH '$.price',* *Quantity integer PATH '$.Quantity') v*

- **Index cardinality issue**: JSON_TABLE() projects out master-detail relationship.
- The *table index* internally creates master-detail relational tables to **hold the relational results** computed by evaluation of JSON_TABLE().
- **Avoid storing repeated:** the column values in the master table are **NOT** repeatedly stored in detail tables for each detailed item.
- **More flexible**: use partial schema to **define index structures** instead of using schema to define base table storage structures.

Schema agnostic

- When there does **not exist any partial schema** in a JSON object collection, or when users can **not anticipate query search patterns** for a collection.

Schema agnostic

- When there does **not exist any partial schema** in a JSON object collection, or when users can **not anticipate query search patterns** for a collection.

- **JSON_EXISTS()** is applied to a JSON object collection to search for **existence of a certain JSON path** with a member value satisfying range criteria.

- Build a JSON inverted index: extends of Oracle's text index to index JSON **object member names**, their **hierarchical relationships** and their **content** leaf data.

- JSON **array elements** are indexed with the **parent array** name containing them.

Schema agnostic

- When there does **not exist any partial schema** in a JSON object collection, or when users can **not anticipate query search patterns** for a collection.

- **JSON_EXISTS()** is applied to a JSON object collection to search for **existence of a certain JSON path** with a member value satisfying range criteria.

- Build a JSON inverted index: extends of Oracle's text index to index JSON **object member names**, their **hierarchical relationships** and their **content** leaf data.

- JSON **array elements** are indexed with the **parent array** name containing them.

- Index both: structures and data found in JSON objects

Let's do it!

Experiment setup

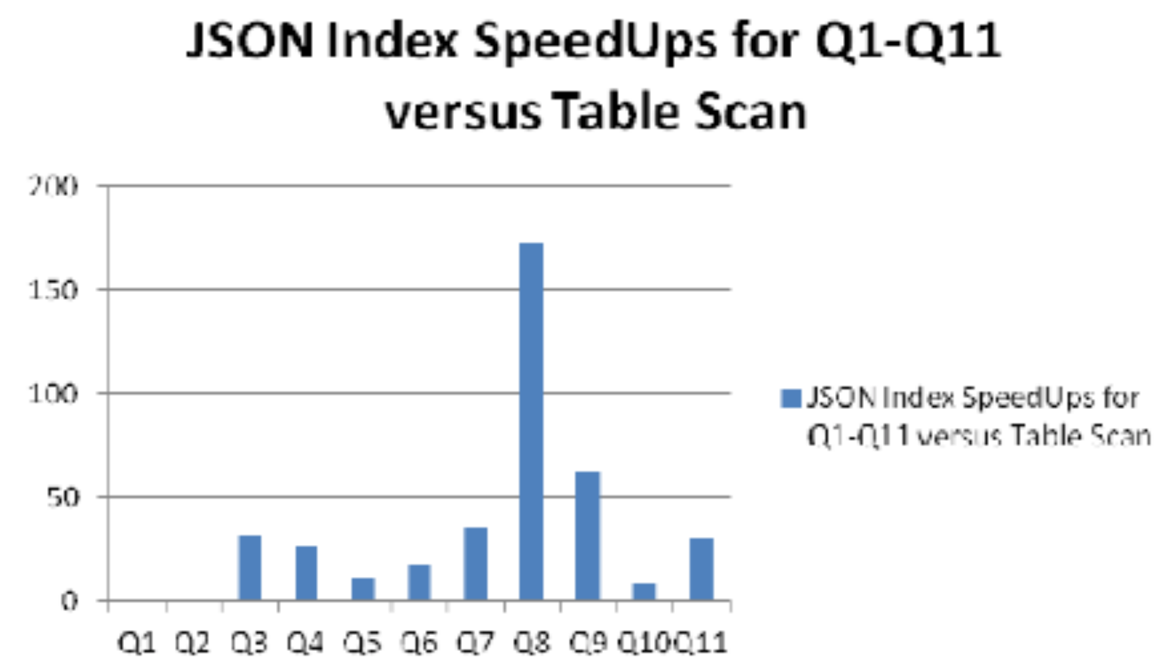
- NOBENCH benchmark data .
- **Oracle RDBMS** (supports SQL/JSON)
- Table NOBENCH_main load the table with JSON object instances.
- **'Aggregated Native JSON Store' approach (ANJS):** functional indices and a JSON inverted index on the column jobj.
- **'Vertical Shredding JSON Store' (VSJS):** Vertical shredding approach of Argo/SQL for JSON objects using path-value relational table.
- PC running Linux kernel 2.6.18 with a 2.53 GHz Intel Xeon CPU and 6GB of main memory.

JSON OBJECT COLLECTION	*CREATE TABLE NOBENCH_MAIN(JOBJ VARCHAR2(4000))*
Functional Index	*create index j_get_str1 on NOBENCH_main(JSON_VALUE(jobj, '$.str1'));* *create index j_get_num on NOBENCH_main(JSON_VALUE(jobj, '$.num' RETURNING NUMBER));* *create index j_get_dyn1 on NOBENCH_main(JSON_VALUE(jobj, '$.dyn1' RETURNING NUMBER));*
JSON Inverted Index	*create index NOBENCH_idx on NOBENCH_main(jobj) indextype is ctxsys.context parameters('json enable ')*

Qry#	SQL/JSON Query
Q1	*SELECT JSON_VALUE(jobj, '$.str1') as str,* *JSON_VALUE(jobj, '$.num' \ RETURNING NUMBER) as num* *FROM nobench_main*
Q2	*SELECT JSON_VALUE(jobj, '$.nested_obj.str') as nested_str,* *JSON_VALUE(jobj, '$.nested_obj.num' RETURNING NUMBER) as nested_num* *FROM nobench_main*
Q3	*SELECT JSON_VALUE(jobj, '$.sparse_000') as sparse_xxx,* *JSON_VALUE(jobj, '$.sparse_009') as sparse_yy0* *FROM nobench_main* *WHERE* *JSON_EXISTS(jobj, '$.sparse_000') AND JSON_EXISTS(jobj, '$.sparse_009'))*
Q4	*SELECT JSON_VALUE(jobj, '$.sparse_800') as sparse_800,* *JSON_VALUE(jobj, '$.sparse_999') as sparse_999* *FROM nobench_main* *WHERE* *JSON_EXISTS(jobj, '$.sparse_800')* *OR* *JSON_EXISTS(jobj, '$.sparse_999'))*
Q5	*SELECT jobj* *FROM nobench_main* *WHERE JSON_VALUE(jobj, '$.str1') = :1*

JSON query performance with and without JSON index

- Queries: with and without indices.
- Q1 and Q2 are queries to project out scalar values from JSON object **without any predicates** used in the WHERE clause so an index can't improve their performance.
- **Functional indices** are used to speed up Q5, Q6, Q7, Q10, Q11 queries for dense static schema.
- A JSON **inverted index** is used to speed up Q3, Q4, Q8, Q9 queries for sparse dynamic schema.

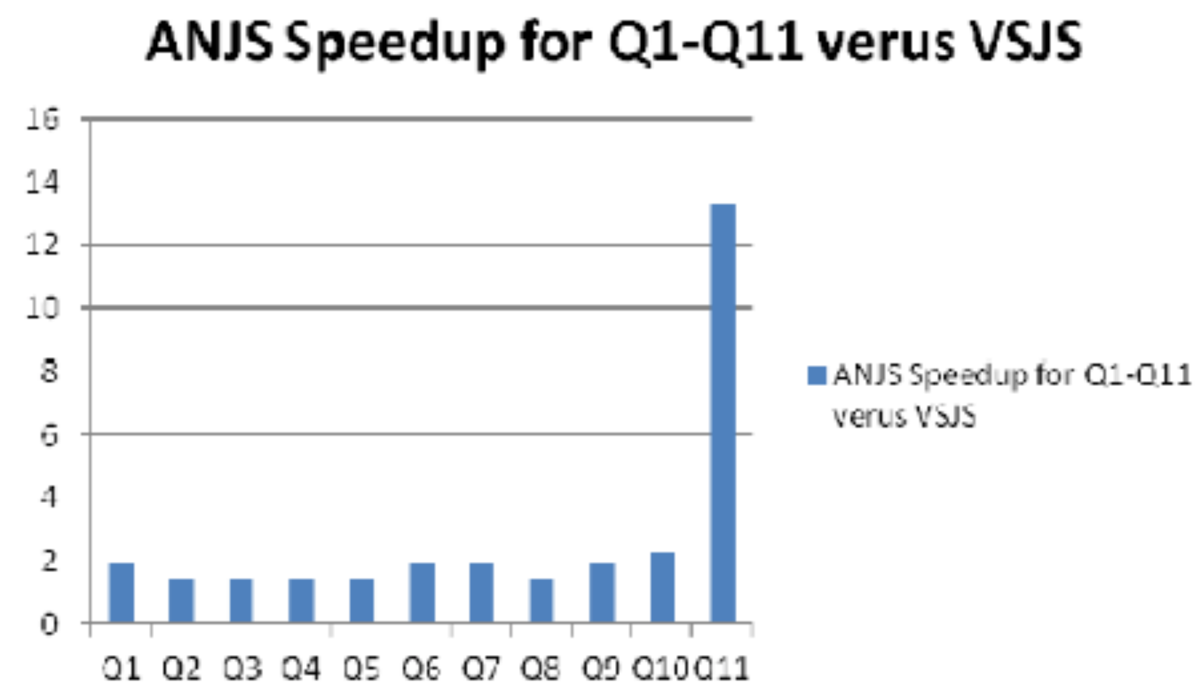


Q1	*SELECT JSON_VALUE(jobj, '$.str1') as str, JSON_VALUE(jobj, '$.num' \ RETURNING NUMBER) as num FROM nobench_main*
Q5	*SELECT jobj FROM nobench_main WHERE JSON_VALUE(jobj, '$.str1') = :1*
Q8	*SELECT jobj FROM nobench_main WHERE JSON_TEXTCONTAINS(jobj, '$.nested_arr', :1)*

Q8: keyword search operation content is embedded inside JSON object members and array elements. JSON full text search needs to incorporate path navigation.

JSON native store approach Versus vertical shredding store approach (1)

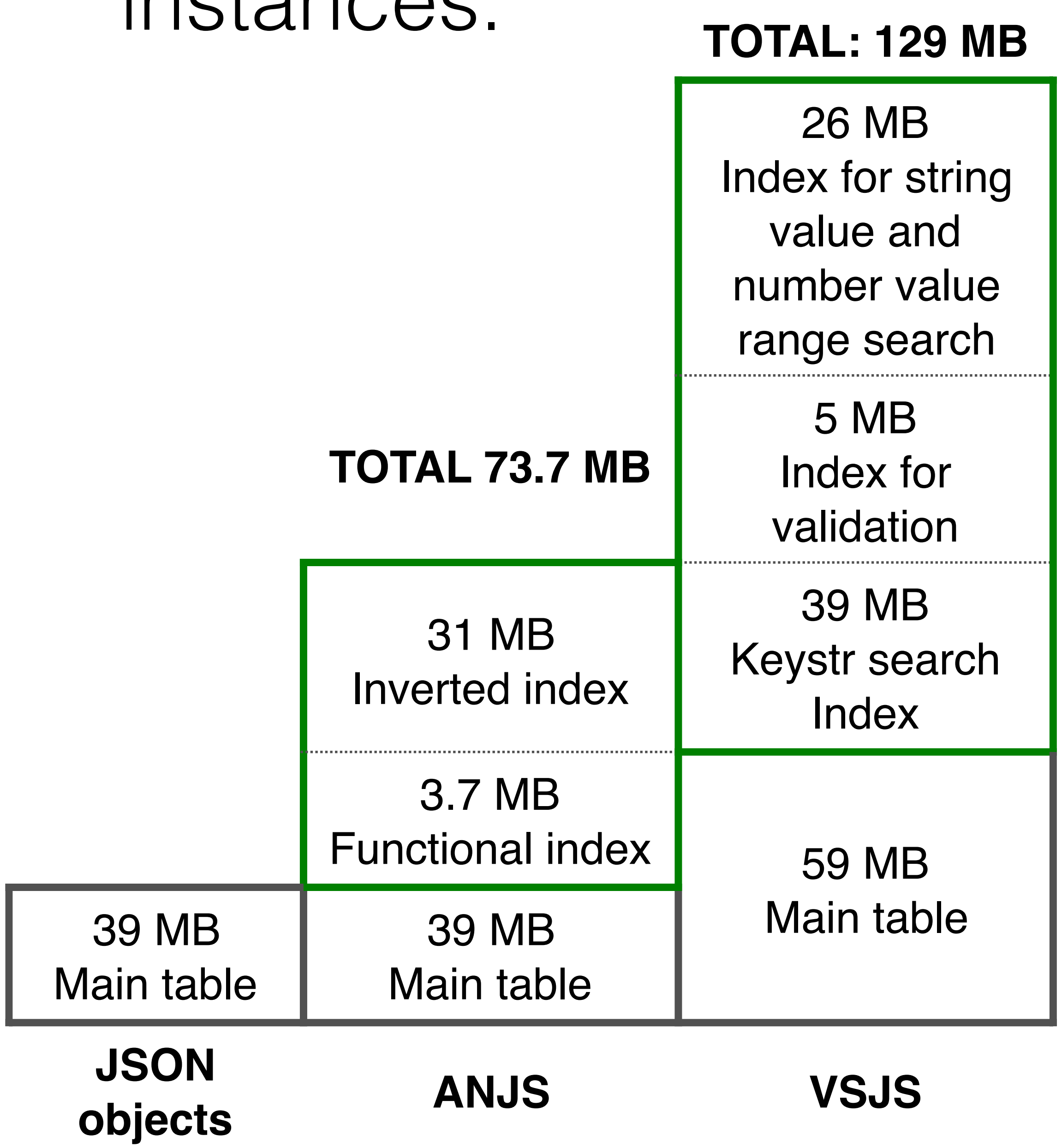
- Argo/3 approach: **vertically shredding** JSON objects and **storing** the resulting **vertical relational table** in the Oracle RDBMS.
- Main path-value relational table: objid, keystr and valstr (indexed by a B+ tree index.)
- An **additional** numeric B+ tree **index** is created on the valstr column for those string values that are valid numbers.
- ANJS with functional and inverted JSON indexes is faster than the VSJS approach.

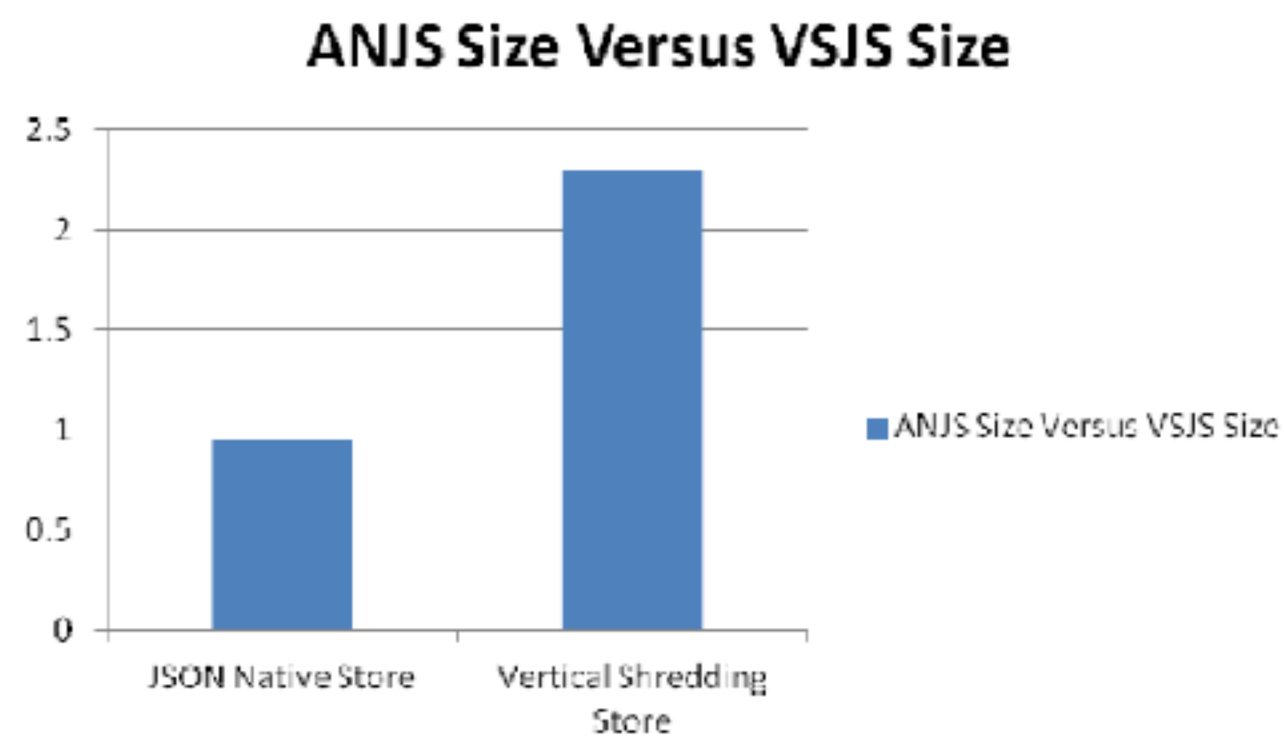


Q11	SELECT left.jobj FROM nobench_main left INNER JOIN nobench_main right ON (JSON_VALUE(left_jobj, '$.nested_obj.str') = JSON_VALUE(right_jobj, '$.str1')) WHERE JSON_VALUE(left_jobj, '$.num' RETURNING NUMBER) BETWEEN :1 AND :2

JSON native store approach Versus vertical shredding store approach (2)

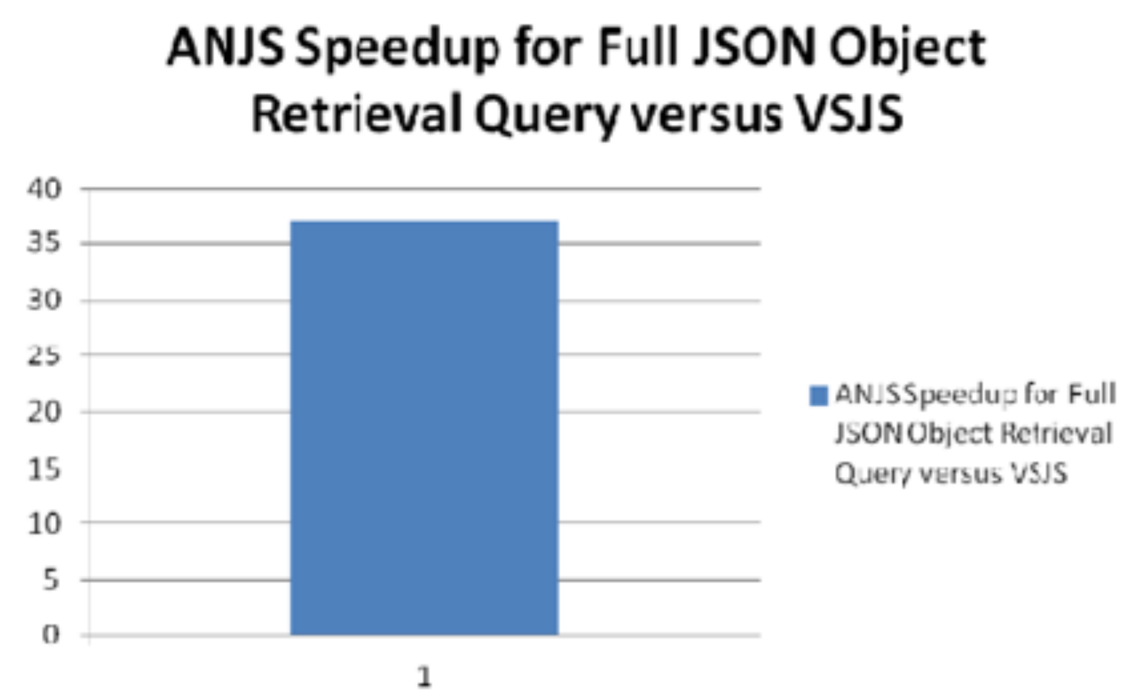
- 50,000 JSON object instances.





JSON native store approach Versus vertical shredding store approach (3)

- JSON store based on vertical shredding = JSON objects not store as a whole.
- **Costly** to respond to common queries that retrieve the whole JSON object as their results.
- The store needs to run **multiple queries to group** all the rows belonging to the same object id and then aggregate all columns of these rows to construct the full JSON object.
- **Difficult object reconstruction:** access many (sometimes un-contiguous) rows when reconstructing matching objects.



Conclusion

- It is necessary the integration of schema-less models within relational database management systems, **BUT** it could be very complex.

Conclusion

- It is necessary the integration of schema-less models within relational database management systems, **BUT** it could be very complex.

So what?

- The three principles approach, allows to improve the results obtained by previous works regarding JSON query and store performance.

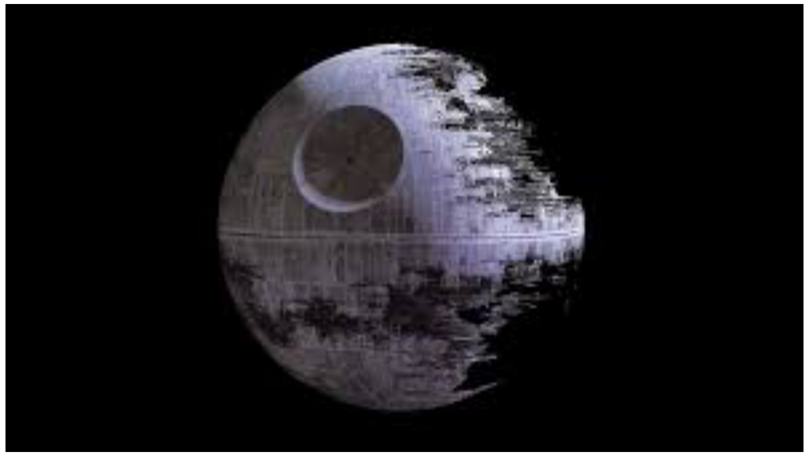
Conclusion

- It is necessary the integration of schema-less models within relational database management systems, **BUT** it could be very complex.

So what?

- The three principles approach, allows to improve the results obtained by previous works regarding JSON query and store performance.

Future work?



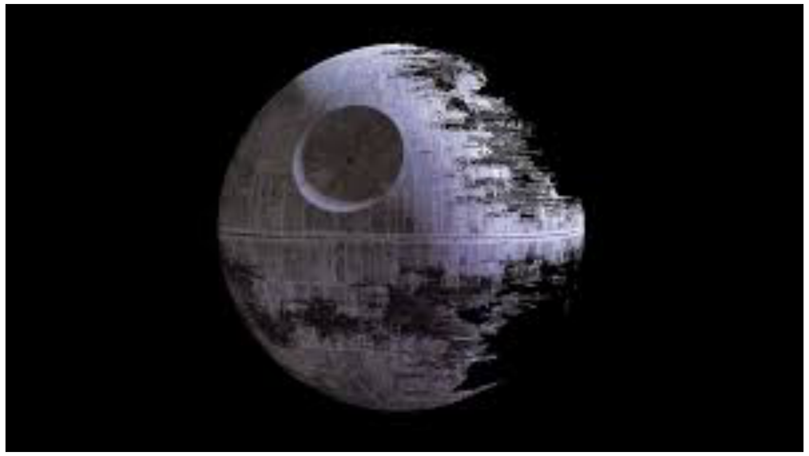
Conclusion

- It is necessary the integration of schema-less models within relational database management systems, **BUT** it could be very complex.

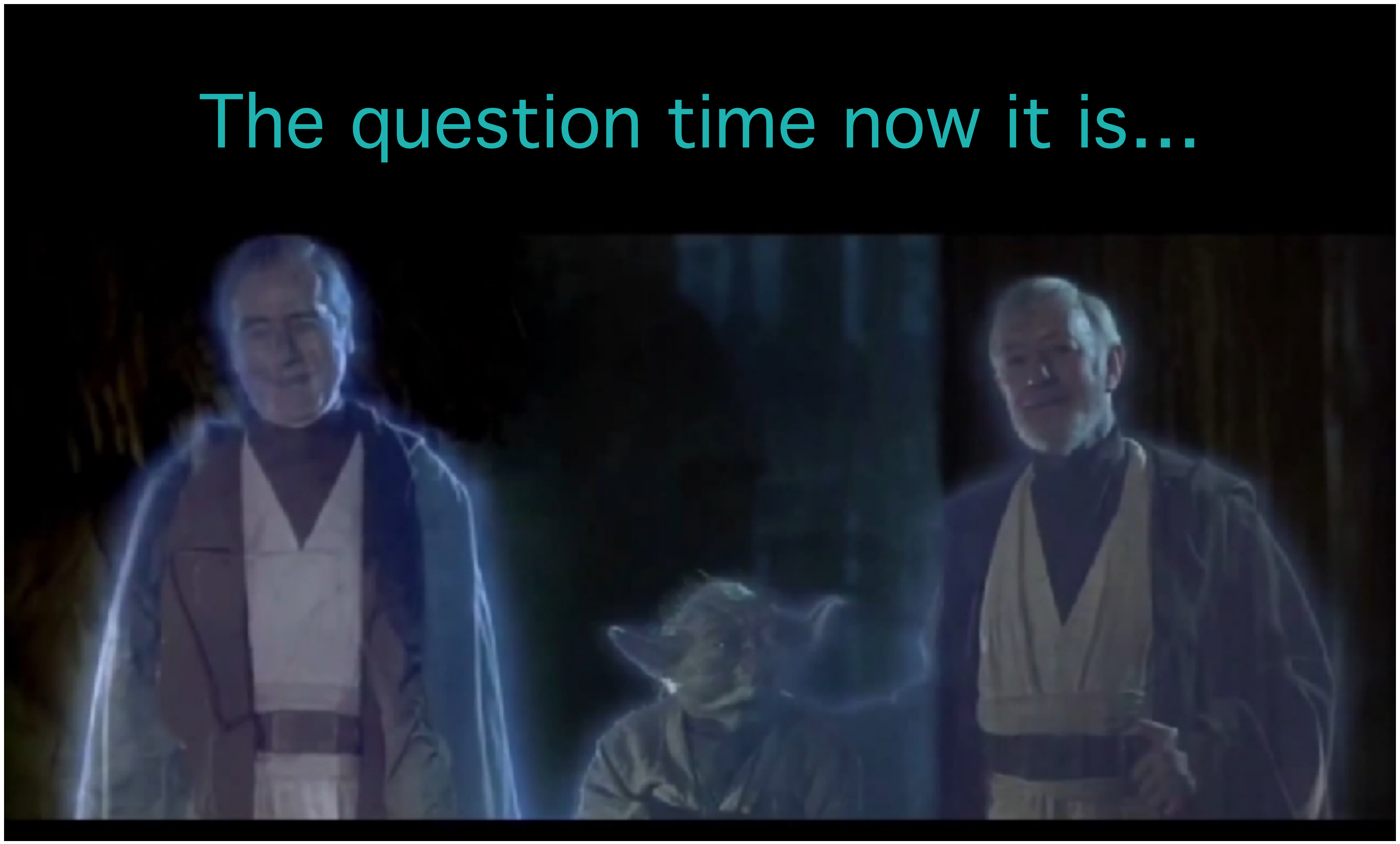
So what?

- The three principles approach, allows to improve the results obtained by previous works regarding JSON query and store performance.

Future work?



- Test the design in other RDBMS, hardware and with different data.
- Try to use similar approach with other NoSQL data models.


The question time now it is…