Faculty of Computer Science
Database and Software Engineering Group

OTTO VON GUERICKE
UNIVERSITÄT
MAGDEBURG

A Gentle Introduction to Document Stores and Querying with the SQL/JSON Path Language

Marcus Pinnecke

Advanced Topics in Databases, 2019/June/7
Otto-von-Guericke University of Magdeburg

Thanks to!

Prof. Dr. Bernhard Seeger & Nikolaus Glombiewski, M.Sc. (University Marburg), and
Prof. Dr. Anika Groß (University Leipzig)

- For their support and slides on NoSQL/Document Store topics

Prof. Dr. Kai-Uwe Sattler (University Ilmenau), and
The SQL-Standardisierungskomitee

- For their pointers to JSON support in the SQL Standard

David Broneske , M.Sc. (University Magdeburg)
Gabriel Campero, M.Sc. (University Magdeburg)

- For feedback and proofreading

About Myself

Marcus Pinnecke, M.Sc. (Computer Science)

- Full-time database research associate
- Information technology system electronics engineer

Faculty of Computer Science
Datenbanken & Software Engineering
Universitätsplatz 2, G29-125
39106, Magdeburg, Germany

About Myself

- /marcus_pinnecke
- /pinnecke
- /in/marcus-pinnecke-459a494a/
- /pers/hd/p/Pinnecke:Marcus
- marcus.pinnecke{at-ovgu}
- /citations?user=wcuhwpwAAAAJ&hl=en
- /profile/Marcus_Pinnecke
- www.pinnecke.info

Rough Outline - What you'll learn

The Case for Semi-Structured Data

- Semi-structured data, arguments and implications
- Overview of database systems, and rankings
- Document Database Model

Document Stores

- Document Stores Overview and Comparison
- CRUD (Create, Read, Update, Delete) Operations in mongoDB and CouchDB

Storage Engine Overview

- Insights into CouchDBs Append-Only storage engine
- Insights into mongoDBs Update-In-Place storage engine
- Physical Record Organization (JSON, UBJSON, BSON, CARBON)

JSON Documents in Rel. Systems

- JSON Support in Relational Database Systems
- SQL/JSON Path Language

Literature & Further Readings (I)

[CBN+07] Eric Chu, Jennifer Beckmann, Jeffrey Naughton,
The Case for a Wide-Table Approach to Manage Sparse Relational Data Sets,
ACM SIGMOD international conference on Management of data. ACM, 2007

[DG-08] Jeffrey Dean, Sanjay Ghemawat
MapReduce: Simplified Data Processing on Large Clusters
Communications of the ACM. ACM, 2008

[MBM+19] Mark Lukas Möller, Nicolas Berton, Meike Klettke, Stefanie Scherzinger, and Uta Störl,
jHound: Large-Scale Profiling of Open JSON Data
BTW 2019, Gesellschaft für Informatik, 2019

[BRS+17] Pierre Bourhis, Juan L Reutter, Fernando Suárez, and Domagoj Vrgoč,
JSON: Data Model, Query Languages and Schema Specification
In Proceedings ACM PODS, pages 123–135, 2017

[SEQ-UEL] Donald D. Chamberlin, Raymond F. Boyce,
SEQUEL: A Structured English Query Language,
Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control, 1974

[PRF+16] Felipe Pezoa, Juan Reutter, Fernando Suarez, Martin Ugarte, and Domagoj Vrgoc,
Foundations of JSON schema,
Proceedings of the 25th International Conference on World Wide Web, 2016

[ISO-SQL] ISO/IEC **Information technology — Database languages — SQL Technical Reports —**
Part 6: SQL support for JavaScript Object Notation (JSON)
http://standards.iso.org/ittf/PubliclyAvailableStandards/c067367_ISO_IEC_TR_19075-6_2017.zip, 2017-03

[SQL-16] Markus Winand,
What's new in SQL:2016
https://modern-sql.com/blog/2017-06/whats-new-in-sql-2016, accessed April 2019

Literature & Further Readings (II)

[JSN-SGA] Douglas Crockford, **The JSON Saga,** https://www.youtube.com/watch?v=-C-JoyNuQJs, accessed April 2019

[WWW-EDP] **European Data Portal,** https://www.europeandataportal.eu, accessed April 2019

[MDB-DOC] **Use Cases - MongoDB**, docs.mongodb.com/ecosystem/use-cases/, accessed March 2019

[MDB-INS] **Insert Documents - MongoDB Manual**, https://docs.mongodb.com/manual/tutorial/insert-documents/, accessed March 2019

[MDB-QRY] **Query Documents - MongoDB Manual**, https://docs.mongodb.com/manual/tutorial/query-documents/, accessed March 2019

[MDB-UPD] **Update Documents - MongoDB Manual,** https://docs.mongodb.com/manual/tutorial/update-documents/, accessed March 2019

[MDB-RMV] **Remove Documents - MongoDB Manual,** https://docs.mongodb.com/v3.2/tutorial/remove-documents/, accessed March 2019

[MDB-RM] **mapReduce - MongoDB Manual,** https://docs.mongodb.com/manual/reference/command/mapReduce/, accessed April 2019

[MDB-TSR] **Text Search - MongoDB Manual,** https://docs.mongodb.com/v3.2/text-search/, accessed April 2019

[MDB-GEO] **Geospatial Queries - MongoDB Manual,** https://docs.mongodb.com/v3.2/geospatial-queries/, accessed April 2019

[MDB-AGG] **Aggregation - MongoDB Manual,** https://docs.mongodb.com/v3.2/aggregation/, accessed April 2019

[CDB-GTS] **Getting Started - Apache CouchDB,** https://docs.couchdb.org/en/stable/intro/tour.html, accessed March 2019

Literature & Further Readings (III)

[CDB-API] **The Core API - Apache CouchDB,**
https://docs.couchdb.org/en/stable/intro/api.html, accessed March 2019

[CDB-REV] **Replication and conflict Model - Apache CouchDB,**
https://docs.couchdb.org/en/stable/replication/conflicts.html#replication-conflicts, accessed April 2019

[CDB-FIND] **1.3.6. /db/_find - Apache CouchDB,**
https://docs.couchdb.org/en/stable/api/database/find.html#selector-syntax, accessed April 2019

[CDB-DSD] **3.1 Design Documents - Apache CouchDB,**
https://docs.couchdb.org/en/stable/ddocs/ddocs.html, accessed April 2019

[CDB-VWS] **4.3.2 Introduction to Views - Apache CouchDB**,
https://docs.couchdb.org/en/stable/ddocs/views/intro.html, accessed April 2019

[SQL-JSN] **JSON data in SQL Server**,
https://docs.microsoft.com/en-us/sql/relational-databases/json/json-data-sql-server?view=sql-server-2017, accessed April 2019

[SQL-JNP] **JSON Path Expression (SQL Server),**
https://docs.microsoft.com/en-us/sql/relational-databases/json/json-path-expressions-sql-server?view=sql-server-2017, April 2019

[RFC-8259] **The JavaScript Object Notation (JSON) Data Interchange Format**,
https://tools.ietf.org/html/rfc8259, accessed March 2019
Request for Comments, Internet Standard, December 2017

[RFC-6901] **JavaScript Object Notation (JSON) Pointer**
https://tools.ietf.org/html/rfc6901, accessed April 2019

[YKB-WTA] **Keith Bostic - WiredTiger [The Databaseology Lectures - CMU Fall 2015]**
https://www.youtube.com/watch?v=GkgDDs9EJUw

Material & References

[MAG] Microsoft Academic Graph / Open Academic Graph
A public available JSON data set of scientific publications metadata.
Used as running example in this lecture.
https://aminer.org/open-academic-graph

[CRBN] Libcarbon and tooling for CARBON files
A C library for creating, modifying and querying
Columnar Binary JSON (Carbon) files.
http://github.com/protolabs/libcarbon

The Document Database Model

The Case for Semi-Structured Data

The Case for Semi-Structured Data (I)

Many arguments for semi-structured data, here two:

Schema is not known in advance, or evolves heavily

1

- Agile methodologies especially for web-services
- Short release cycles, incremental improving systems
- Operating on third-party datasets, analysis
- …

Database normalization is not required, or optional

2

- Scale-out performance by redundancy and decoupling
- Hierarchical records to avoid effort for “joining”
- …

OTTO VON GUERICKE UNIVERSITÄT MAGDEBURG

Schema Considerations

The Case for Semi-Structured Data (IV)

Schema is not known in advance, or evolves heavily

- **Def (schema)** A schema describes structure of entities/records belonging to a class or group (e.g., a table)
  - Description of mandatory/optional fields and data types, maybe ordering
  - Determines record identity (i.e., primary keys) and references (i.e., foreign keys)
  - Often used to express constraints on records, potentially spanning multiple tables
  - Typically used by the system for (physical query) optimization
- A schema is user-defined and database-specific
  - The system is *not* allowed to expose a semantic-inequivalent, inconsistent schema
  - Internal modifications on the schema are possible, though
    - Don't allocate storage for columns only containing *null* values
    - Reduce memory footprint by minimizing number of bytes for field types
    - Denormalize multiple tables to one "Wide Table" [CBN+07]
    - ...

The Case for Semi-Structured Data (V)

Schema is not known in advance, or evolves heavily

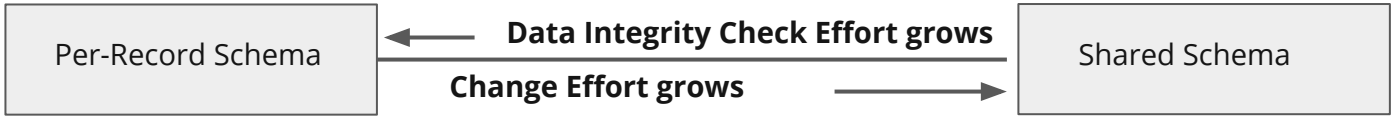
- System must react to change requests on the schema
  - Typically, a system becomes
    - Slower (and saves resources), or
    - Consumes more resources (and is still fast)

    the more actions are required to apply a change in a schema:
    - Potentially undo internal modifications
    - Re-evaluate decisions on storage optimization
  - In addition, complexity depends on
    - the number of
      - records that must be re-written
      - groups/tables that must be locked
      - the degree of normalization
    - on the complexity of constraints
    - on effort to rebuild indexes
    - ...

The Case for Semi-Structured Data (VI)

Schema is not known in advance, or evolves heavily

- **Trade-Off** between control over groups of records at once *vs* fine-grained flexibility per record



  - At which granularity shall **schema-flexibility** be applied? The more fine-grained, the less effort is needed to change the schema of single records.
    - **Wide-Tables** All records (i.e., single-table-database schema)
    - **Relational Systems** Groups of records (i.e., per-table schema)
    - **NoSQL Systems** Single records (i.e., per-record-schema)
  - At which granularity is **data integrity** (esp. schema-match) checked? The more records are bundled in groups with a shared schema, the less effort is needed to perform such checks.

The Case for Semi-Structured Data (VII)

Schema is not known in advance, or evolves heavily

Consequence An `ALTER TABLE T` statement in a productive environment may be cumbersome *if* the system is built for structured (tabular) data with a (assumed mostly static) schema on tables

- *All* records inside *T* are affected by the change
- Cascading deletes/updates in other tables may occur (cf., normalization)

Normalization Considerations

The Case for Semi-Structured Data (VIII)

Data normalization is not required, or optional

- **Def (normalization)** Database normalization is a systematic process in (relational) database design to eliminate data redundancy and improve data integrity by reorganizing tables via column-splits into new tables.

- **Goal** making data dependencies explicit for enabling data integrity checks.

Without database normalization there is the high risk of **database anomalies**

- **Semi-structured data is typically not normalized**

The Case for Semi-Structured Data (IX)

Data normalization is not required, or optional

- **Def (data redundancy)** Data redundancy is the existence of (full/partial) copies of an actual datum (e.g, a field value) making the information redundant (i.e., information is given n times, and n-1 times can be removed w/o information loss)
  - Pros
    - Robustness Recover from corruption or data loss (“use the copy instead”)
    - Performance No need to grab a datum from its original location
  - Cons
    - Storage Costs Additional space is needed needed for copies
    - Inconsistency Update on one copy may not be reflected in others
    - Data corruption No data integrity

The Case for Semi-Structured Data (X)

Data normalization is not required, or optional

- Data integrity is a property that refers to the quality of data w.r.t.
  - accuracy and consistency

  and is validated over the entire lifespan of a datum.
  - Pros
    - Data is not modified unintentionally
  - Cons
    - Requires effort for validation and/or database design (via normalization)

There is almost no reason not to aim for data integrity, i.e., you want consistent data

Keep in mind that data integrity is related to ACID transactions and its granularity.

The Case for Semi-Structured Data

Semi-structured data is reasonable *if* an application scenario implies/requires

- Limited Domain Knowledge — Proper schema can't be determined upfront/changes anyway
- Efficient Schema-Evolution — Fast structural changes on single records (add/remove fields)
- Robust Performance First — Storage costs, consistency, and (strong) integrity secondary

Use cases (by example of MongoDB) [MDB-DOC]

- Operational Intelligence (Storing Log Data, Hierarchical Aggregation)
- Product Data Management (Product Catalog, Inventory Management, Category Hierarchy)
- Content Management Systems (Metadata and Asset Management, Storing Comments)

The Case for Semi-Structured Data

How often is it the case?

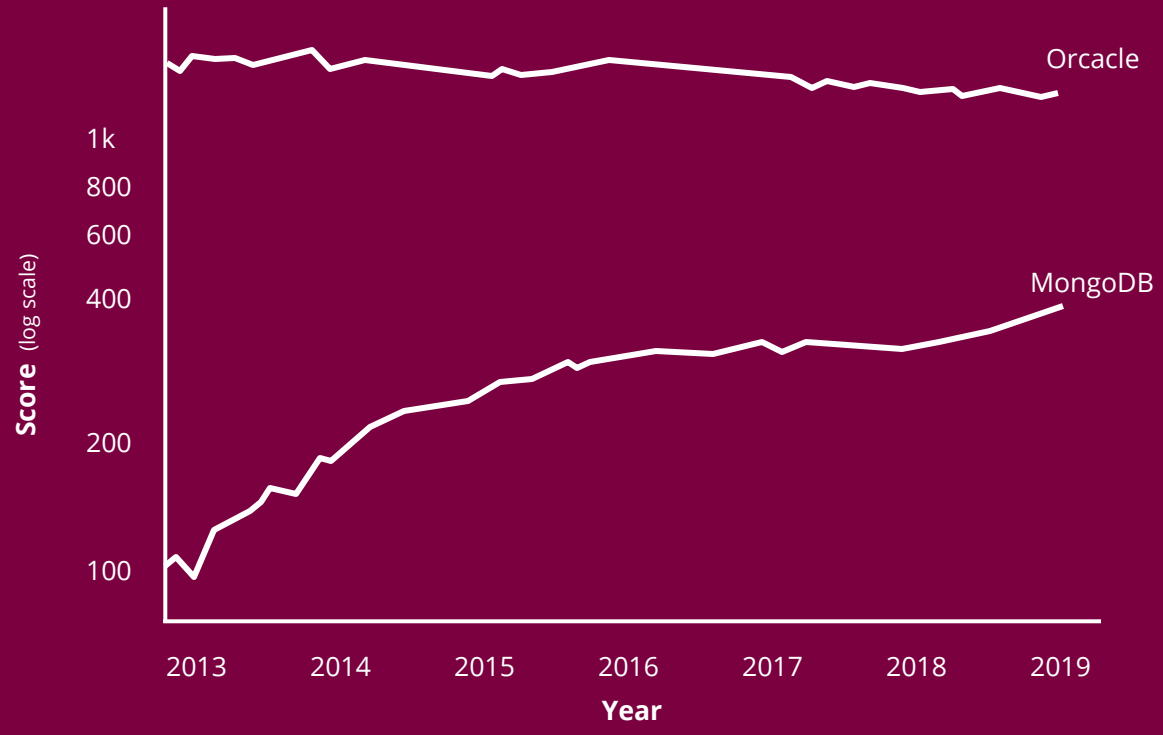
Rank	Database System Name	Data Model
1	Orcale	Relational, Multi
2	MySQL	Relational, Multi
3	SQL Server	Relational, Multi
4	PostgreSQL	Relational, Multi
5	MongoDB	Document Model

Source https://db-engines.com/en/ranking/ (last update march 2019)

The Case for Semi-Structured Data
How often is it the case?

Notes

- A document model system is in top 5 of db-engines ranking
- Best (Oracle) has still 3x the scope value of MongoDB
- MongoDB has a better ranking trend, though



Source https://db-engines.com/en/ranking/ (last update march 2019)

The Case for Semi-Structured Data

Which document store systems to know?

Rank	Database System Name	Score
1	MongoDB	401.34
2	Amazon DynamoDB	54.49
3	Couchbase	33.80
4	Microsoft Cosmos DB	24.83
5	CouchDB	18.63

Source https://db-engines.com/en/ranking/ (last update march 2019)

OTTO VON GUERICKE UNIVERSITÄT MAGDEBURG

Semi-Structured Data

Document Database Model (I)

Documents A record (called *Document*) in a document store is typically:

- Semi-structured per-record schema
- Denormalized contains redundant data
- Potentially nested may contain other records
- Self-Identifiable no user-def. primary key (system-generated *object id* `_id` instead)
- Self-Contained no foreign keys to refer to other records

Collections Similar records are organized in groups (typically called *Collections* or *Database*):

- Records of similar but not necessarily equal schema and purpose
- No constraints enforced by the database (instead user-empowerment)

Document Database Model (II)

Comparison Collection of documents vs table of tuples (by example of [MAG], excerpt)

title (string)	**n_citations**	**authors** (object array)			**references** (string array)	
		(idx)	**name** (string)	**org** (string)	(idx)	(value)
Structural defects in GaN	(not in list)	0	S. Ruvimov	Div. of Mater. Sci (...)	0	07d52a00-109f(...)
					1	48f2de10-2c83(...)
		1	Z. Liliental-Weber	(not in list)	...	...
					5	df0e1313-9b65(...)
A decision support tool	50	0	Charles White	(not in list)	(not in list)	

A document is (typically) structured similar to a **JSON** document.

Document Database Model (III)

Comparison Collection of documents vs table of tuples (by example of [MAG], excerpt)

JSON

```
[
   {
      "title":"Structural defects in GaN",
      "authors":[
         { "name":"S. Ruvimov", "org":"Div. of Mater. Sci (...)" },
         { "name":"Z. Liliental-Weber" }
      ],
      "references":[
         "07d52a00-109f(...)", "48f2de10-2c83(...)", "6d1efe54-c7aa(...)",
         "c2950b99-d734(...)", "ccab2fc4-276d(...)", "df0e1313-9b65(...)"
      ]
   },
   {
      "title":"A decision support tool",
      "n_citations": 50,
      "authors":[
         {  "name":"Charles White" }
      ]
   }
]
```

JavaScript Object Notation (I)

What is JavaScript Object Notation (JSON) Data Interchange Format **not** [json.org/json.pdf]

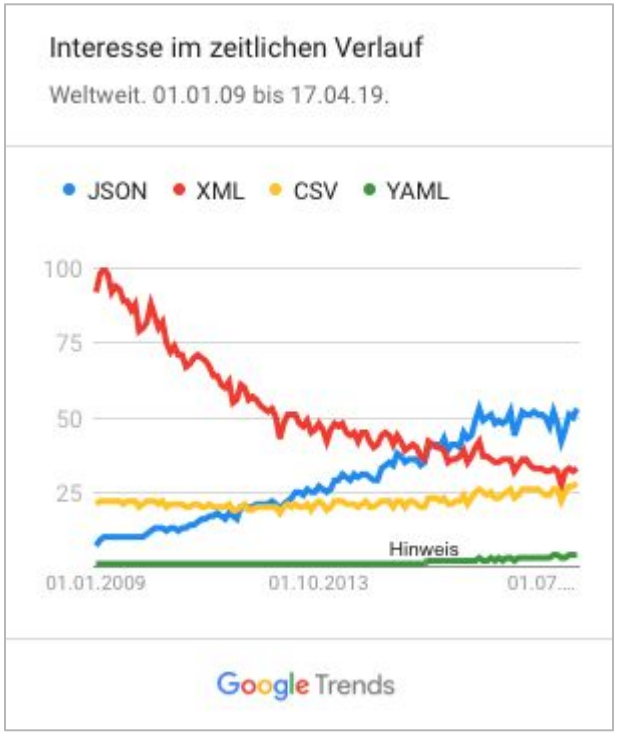
- JSON is **not** a document format (like `.docx` of Microsoft Word)
- JSON is **not** a markup language (like `.xml`)
- JSON is **not** a general serialization format (i.e., JavaScript ≠ JSON)
  - No cyclical/recurring structures
  - No invisible structures
  - No functions

JSON is a data interchange format (like RDF, XML, YAML, CSV,...)

JavaScript Object Notation (II)

What is JavaScript Object Notation (JSON) Data Interchange Format

- rooted back to early usage in Netscape (1996) [JSN-SGA]
- Designed for applications that do not have specific knowledge of contained data
  - internet/network applications and transfer:
    - REST (**Re**presentational **s**tate **t**ransfer)-API call results
    - AJAX (**a**synchronous **J**avaScript **a**nd **X**ML) requests
  - open datasets among several domains [WWW-EDP]:
    - Energy & Transport
    - Regions & Cities
    - Economy & Finance
    - Government & Public Sector
    - Justice, Legal System & Public Safety
    - ....
- Well described in Request-for-Comments 8259 [RFC-8259]
- Formal model of JSON in 2017 by Bourhis et al. [BRS+17]
- Currently, most interesting one among alternatives
  - XML, CSV, or YAML



JavaScript Object Notation (III)

What is JavaScript Object Notation (JSON) Data Interchange Format [RFC-8259]

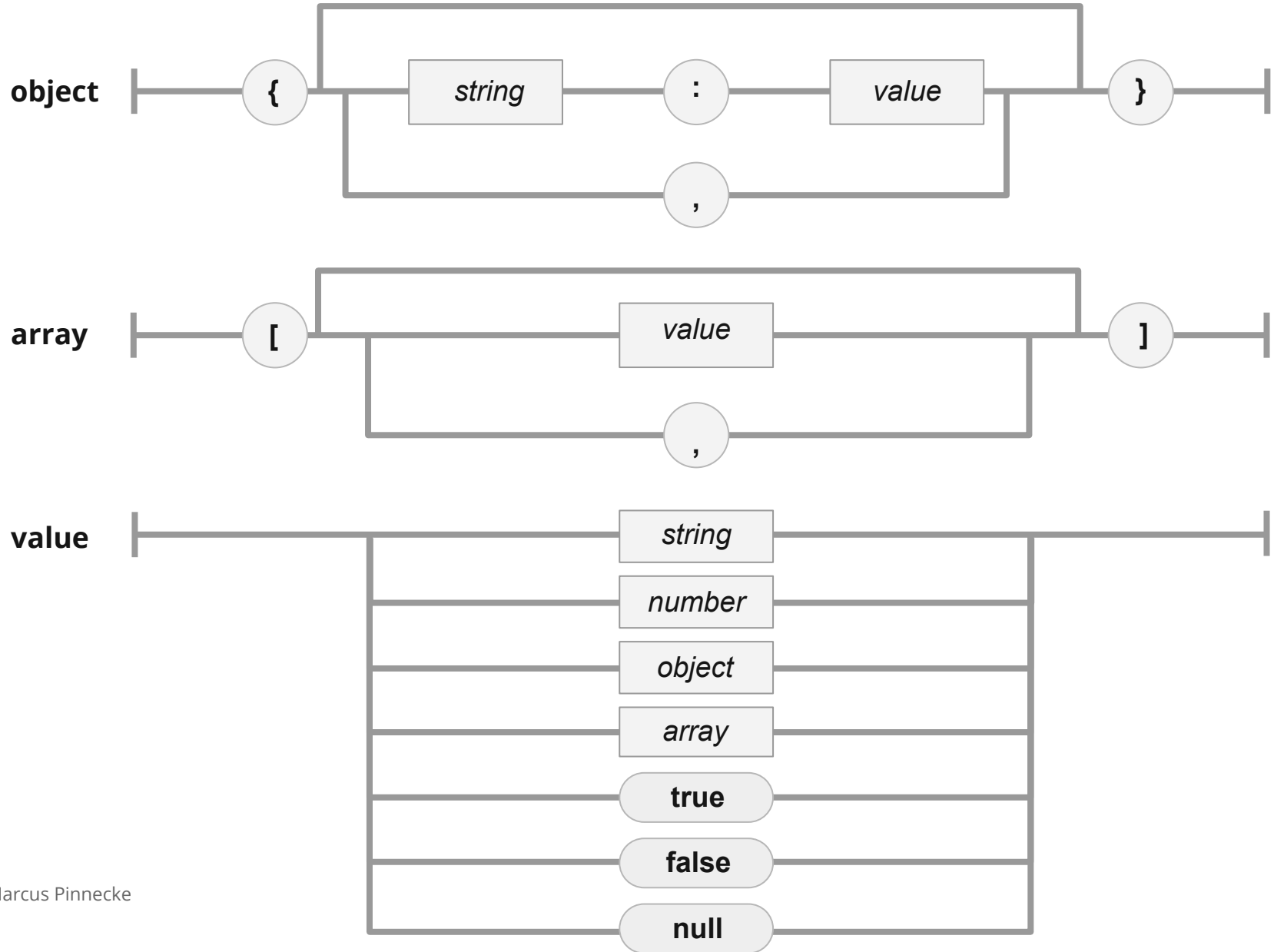
- Lightweight, language-independent data interchange format
  - formatting rules for the portable representation of structured data
  - human-readable format, text-based (file extension `.json`)
  - Internet Media (MIME) type for JSON is `application/json`
  - associated with the JavaScript programming language
- Represented data types
  - primitive (strings, numbers, booleans, and null)
  - structural (objects, and arrays)

JavaScript Object Notation (IV)

What is JavaScript Object Notation (JSON) Data Interchange Format [RFC-8259]

- Building blocks
  - **Object** (potentially empty) unordered collection of *properties* (key-value pairs):
    - key is a string
    - value is a string, number, boolean, `null`, object, or array
  - **Array** (potentially empty) ordered sequence of values
    - primitive values (strings, numbers, booleans)
    - compound values (object, array)
    - literals (`true`, `false`, and `null`)

JSON Syntax Diagram (simplified)



JSON Schema

JSON Schema

No mechanism provided in JSON Spec for verification against a particular schema

- "JSON is self-describing": syntax check only according JSON Spec [RFC-8259]
- Without schema to validate against, a lot of cases must be considered
  - "`n_citations`" field (number of citations) in [MAG] is formatted as number or as string
    - Requires type conversions
  - "`id`" field to identify a publication in [MAG]; does it exist in all 100+ Mio documents?
    - Requires existence checks
  - ...
- Efforts for schema validation called **JSON Schema** [PRF+16]
  - schema language to constrain the structure and to verifying the integrity
    - string values with *min*/*max* number of characters or matching *regex* pattern
    - constraining fields being *not*/*allOf*/*anyOf* type
    - constraining fields having a value out of a predefined set
  - So far, less interest in internet community to support schemata

JSON Pointer

JSON Pointers

Syntax to refer to specific value within a JSON document [RFC-6901]

JSON

```
{  "title":"Structural defects in GaN",
   "authors":[
      { "name":"S. Ruvimov", "org":"Div. of Mater. Sci (...)" },
      { "name":"Z. Liliental-Weber" }
   ] }
```

- A JSON pointer is a string of reference tokens, each prefixed by a `/`
  - Evaluation starts with reference to root value
  - Completes with some value within the document
  - Reference tokens are evaluated sequentially
    - If value is JSON object, new reference value is property with reference token as key
      - Key name is equal to reference token by case-sensitive string equality
    - If value is array, reference token must contain
      - zero-based index *i* to refer to *i*-th element in array

JSON Pointer

```
""                    (entire document)
"/title"              "Structural defects in GaN"
"/authors"            [ { ... }, { ... } ]
"/authors/0"          { "name":"S. Ruvimov", "org":"Div. of Mater. Sci (...)" }
"/authors/0/name"     "S. Ruvimov"
```

Summary

The Case for Semi-Structured Data

Summary The Case for Semi-Structured Data

Semi-structured data, arguments and implications

- Schema is not known in advance, or evolve heavily
- Database normalization is not required, or optional
- Application scenarios and use cases

Overview of database systems, and rankings

- Top-5 data models & trends
- Top-5 document stores

Document Database Model

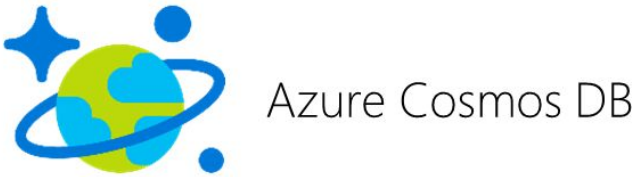
- Fundamental terms (document, collection)
- Document collection vs tuples in tables
- JavaScript Object Notation (JSON): scoping, history, syntax
- JSON Schema to verify a document against a schema
- JSON Pointer to refer to specific value within a document

OTTO VON GUERICKE UNIVERSITÄT MAGDEBURG

Document Stores

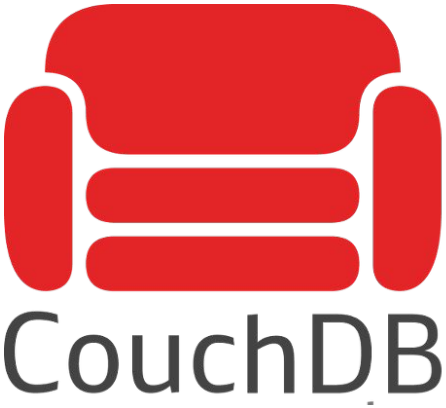
(User Land)

Document Stores

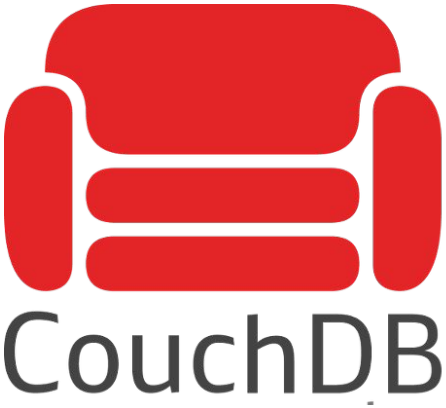


Couchbase

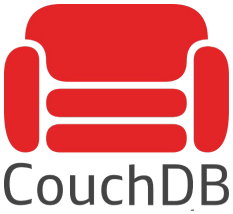
(...)



Document Stores



Document Stores in Comparison



- **Append-Only Storage**
  - Multi Version Concurrency Control (MVCC)
  - Availability over consistency
  - Master-Master Architecture
    - every instance is a master
    - sync via merge-replication
    - eventual consistency
  - **Records**: JSON, database of records
- **Queries** via REST, and views (map-reduce)
- **Communication** via REST API
  - **in** `curl -X GET http://127.0.0.1:5984/mydb/42`
  - **out** `{ "_id": "42", "_rev": "1-3(...)", ...} }`

- **Update-In-Place Storage (WiredTiger)**
  - Optimistic Concurrency Control (Document-Level)
  - MVCC (Snapshots & Checkpoints)
  - Consistency over availability
  - Sharding Architecture
    - instances are partitions of database
    - union of partitions is logical database
    - strong consistency
  - **Record**: BSON, database of records collections
- **Queries** via JavaScript, and map-reduce
- **Communication** language-embedded driver
  - **in** `db.mydb.find({"_id" : ObjectId("42")})`
  - **out** `{ "_id": "42", ...} }`

CRUD Operations in Document Stores

CRUD Operations

Create, Read, Update, and Delete

(In a Nutshell)

CRUD Operations

Create, **R**ead, **U**pdate, and **D**elete

- **Create** Inserts new documents to a collection [MDB-INS]
  - *insertOne* to insert a single document
  - *insertMany* to insert multiple documents at once

```JavaScript
db.academicGraph.insertOne(
   {
      "title":"A decision support tool",
      "authors":[
         {  "name":"Charles White" }
      ]
   }
)
```

Inserts a document with *fields* `title` and `authors`, and *values* `A decision ...` resp. an object array to collection `academicGraph`.

```JavaScript
db.academicGraph.insertMany(
   D1, D2, ... , Dn
)
```

Similar

CRUD Operations

Create, **R**ead, **U**pdate, and **D**elete

- **Create** Inserts new documents to a collection [MDB-INS]
  - *insertOne* to insert a single document
  - *insertMany* to insert multiple documents at once

  The following semantic is applied
  - The collection (e.g., `academicGraph`) is created if not already present
  - Each document D_1, D_2, . . . , D_n gets a unique object id (`_id` field) assigned (see later)
  - A single document write is an atomic operation

CRUD Operations

Create, **R**ead, **U**pdate, and **D**elete

- **Read** Returns documents from a collection based on a query condition [MDB-QRY]

```JavaScript
db.academicGraph.find( dot-notated-query-filter-document )
```

  - **Query Filter Document** is a document that specifies query conditions with mixture of *exact match* and *query operator* expressions.
  - **Dot-Notation** is used to specify *array* elements (by index), or *fields* of *nested documents.*

CRUD Operations

Create, **Read**, **U**pdate, and **D**elete

- **Exact match** selects documents having all fields as provided

```JavaScript
{ field: value, … }
```

  - *field* key name
  - *value* exact value to match

  In case multiple such pairs are provided they are in conjunction (AND)

Example

```JSON
{ "title":"A decision support tool",
  "authors":[
      {  "name":"Charles White" }
  ] }
```

Exact Match

```
in  { "title":"A decision support tool" }
out { "title": /* … */, "authors":[ {  /* … */ } ] }
```

Exact Match

```
in  { "title":"A decision support tool","citation": 5 }
out (none)
```

CRUD Operations mongoDB

Create, **Read**, **U**pdate, and **D**elete

- **Query operator** evaluates expression and selects/projects documents

```JavaScript
{ field: { operator: value }, …}
```

  - *field* key name
  - *value* object with *operator* and *value*
    - Operators are not enquoted and start with $, e.g., $ne for *not equal to*
    - Selection
      - Comparison (not equal to, less than,…) & Logical (and, not, nor, or)
      - Element (have at least that field, have specific value type)
      - Evaluation (aggregation, modulo, regex,…)
      - Geospatial (intersection, within, near,…)
      - Array (all elements contained, array length is,…)
      - Bitwise operations and comment
    - Projection
      - (First element in array that matches, score values, offset/limit,…)

CRUD Operations

Create, **Read**, **U**pdate, and **D**elete

- **Dot-Notation** is used to specify array elements (by index) *or* to access a nested field

```JavaScript
array-field.index
```

- *array-field* is key name of an array property
- index is zero-based element index to consider

```JavaScript
field.nested-field
```

- *field* key name
- nested-field key name

Example

```JSON
{ "title":"A decision support tool",
  "authors":[
     {  "name":"Charles White" }
  ] }
```

Array Access — Dot Notation & Result

```
authors.0
{ "name":"Charles White" }
```

Nested Field (via Array) — Dot Notation & Result

```
authors.0.name
"Charles White"
```

CRUD Operations

Create, **Read**, **U**pdate, and **D**elete

- **Read** Query for aggregations [MDB-AGG]
  - MongoDB supports three aggregation processes
    - *Aggregation Pipeline* — flexible multi-stage data processing framework (filters,grouping, sorting, aggregation, transformation,...)
    - *Single Purpose Operations* — three specialized operations (count, group, duplicate elimination)
    - *MapReduce* — (see later)

CRUD Operations

Create, **Read**, **U**pdate, and **D**elete

- **Read** Query for aggregations [MDB-AGG]
  - MongoDB supports three aggregation processes
    - *Aggregation Pipeline* flexible multi-stage data processing framework (filters,grouping, sorting, aggregation, transformation,...)

CRUD Operations

Create, **Read**, **U**pdate, and **D**elete

- **There is more for read operations!**
  - *Text search* via a `$text` operator and dedicated index, see [MDB-TSR]
  - *Geospatial queries* over GeoJSON and dedicated index, see [MDB-GEO]
  - ...

CRUD Operations mongoDB

Create, **R**ead, **Update**, and **D**elete

- **Update** Modifies documents matching a condition [MDB-UPD]

```JavaScript
db.academicGraph.updateOne( filter, update, options )
db.academicGraph.updateMany( filter, update, options )
db.academicGraph.replaceOne( filter, update, options )
```

  - `filter` document w/ selection criteria (dot-notated query filter document, see `find`)
  - `update` document w/ update statements, containing *update operators*
    - *Field updates* set to *x* (if less/greater *y*), inc by *x*, rename/delete field,...
    - *Array updates* first/all/some element(s) only, add/remove value,...
    - *Modifications* add multiple values to array, set element at, slices, sort,...
    - *Bitwise* performs bitwise *AND*, *OR*, *XOR* on integer values
  - `options` document w/ update options
    - add new document if no match (***upsert***), require update in at least *x* replicas/shards, string compare options (e.g., locale or case-sensitivity), condition on array elements to update “some” elements

CRUD Operations mongoDB

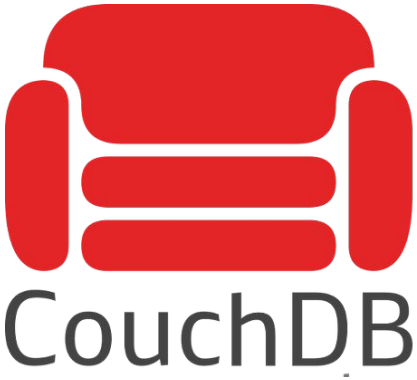
Create, **R**ead, **U**pdate, and **Delete**

- **Delete** Deletes documents matching a condition [MDB-RMV]
  - *deleteOne* to delete a single document
  - *deleteMany* to delete multiple documents at once

(Similar to *find*)

CRUD Operations

Create, Read, Update, and Delete



(In a Nutshell)

CRUD Operations

Create, **R**ead, **U**pdate, and **D**elete

- **Create** Inserts new database *academic_graph* [CDB-GTS]

```bash
$ curl -X PUT http://127.0.0.1:5984/ academic_graph
```

```json
{"ok": true}
```

HTTP **PUT** method used on CouchDB URI to insert new database (if not exists) via URL-encoding

Note: CouchDB URI is deployment-dependent (here: port 5984 on localhost)

CRUD Operations

Create, **R**ead, **U**pdate, and **D**elete

- **Create** Inserts new document to database *academic_graph* [CDB-API]

Bash

```
curl -X PUT http://127.0.0.1:5984/academic_graph/<primary-key> -d \
     '{ "title":"A decision support tool", "authors":[ {      \
     "name":"Charles White" } ] }'
```

JSON

```
{"ok":true,"id":"<primary-key>","rev":"1-2902191555"}
```

(rev: revision; see later)

HTTP **PUT** method mit parameter **-d** to insert new document with id *primary-key*

- <primary-key> user-defined (unique) identifier for document
  - dataset-dependent, such as paper's "id" in MS academic graph
  - user-defined and automatically generated externally
  - system-defined by calling `curl -X GET http://127.0.0.1:5984/_uuids`
- -d curl-dependent parameter to use remainder as body text for request
- '{ … }' document content to be inserted

CRUD Operations

Create, **Read**, **U**pdate, and **D**elete

- **Read** Lists all installed databases [CDB-GTS]

```
$ curl -X GET http://127.0.0.1:5984/ _all_dbs
```

Bash

```
["acadmic_graph"]
```

JSON

HTTP **GET** method on pre-defined point *_all_dbs* to receive all databases

CRUD Operations

Create, **Read**, **U**pdate, and **D**elete

- **Read** Retrieve a particular document by its id [CDB-API]

Bash
```
$ curl -X GET http://127.0.0.1:5984/academic_graph/<primary-key>
```

JSON
```
{"_id":"<primary-key>","_rev":"1-2902191555", "title":"...",  \
"authors":[ { ... } ]}
```

HTTP **GET** method on primary-key (document-id) in database
Results in inserted document with two new field

- `_id` the primary-key assigned to the document
- `_rev` the revision number of the returned document content

CRUD Operations

Create, **Read**, **U**pdate, and **D**elete

- **Read** Returns documents from a collection based on a query condition [CDB-FIND]

Bash

```
$ curl -X POST http://127.0.0.1:5984/academic_graph/_find

{
     "selector": { ... } JSON object describing query condition
     "limit":    N       Maximum number of results
     "skip":     M       Offset first M results entries
     "sort":     [ ... ] JSON object array describing sort policy
     "fields":   [ ... ] String array to define field projection
     Other descriptors for further options
}
```

CRUD Operations

Create, **Read**, **U**pdate, and **D**elete

- **Read** Returns documents from a collection based on a query condition [CDB-FIND]
  - Query predicate (required)

```Bash
"selector": {
     "<field-name>": <value>,
     ...
}
```

  - Restricts the result set to documents having the field `field-name` with exactly the value `value` (implicit `$eq` operator). In case of multiple such pairs, the logical **AND** is applied (implicit `$and` operator).
  - Nested fields can be restricted by
    - nested values: `"<field-name>": { <nested-field-name>: <value> }`
    - dot-notation values: `"<field-name>.<nested-field-name>": <value> }`

CRUD Operations

Create, **Read**, **U**pdate, and **D**elete

- **Read** Returns documents from a collection based on a query condition [CDB-FIND]
  - Query predicate (required)
  - More complex queries can contain (explicit) operators

```
"<field-name>": { "$<operator>": <arguments> }
```

    - *Combination*
      - `$and, $or, $not, $nor, $all, $elemMatch, $allMatch`
    - *Condition*
      - Comparison `$lt, $lte, $eq, $ne, $gte, $gt`
      - Existence `$exists, $type`
      - Array `$in, $nin, $size`
      - Misc `$mod, $regex`

CRUD Operations

Create, **Read**, **U**pdate, and **D**elete

- **Read** Returns documents from a collection based on a query condition [CDB-FIND]
  - Ordered By (optional)

```
"sort": [
     {"<field-name>": ("asc"|"desc")},
     ...
]
```

JSON

  - States a list of objects for which the result should be ordered, each containing
    - a `field-name` to specify the field
    - a sort direction (ascending, descending)

CRUD Operations

Create, **Read**, **U**pdate, and **D**elete

- **Read** Returns documents from a collection based on a query condition [CDB-FIND]
  - Projection (optional)

```JSON
"fields": [ "<field-name>",... ]
```

  - If given, projects the result set to field names provided in the array
  - Implicit (internal) fields must be explicitly added, if projection is applied:
    - revision field (`"_rev"`)
    - document id field (`"_id"`)

CRUD Operations

Create, **Read**, **U**pdate, and **D**elete

- **Read** Query for aggregations and the Design Document concept [CDB-DSD]
  - *Design Documents* REST API endpoints running user-defined (JavaScript) code
    - *Views* Querying and Aggregation w/ **MapReduce** (see later)
      - Each view is managed in its own B^+-tree
      - All views of same document are in same index
    - *Show (List)* Document formatting (on view results)
    - *Update* Client-defined modification stored procedures
    - *Filter* Stream processing of change feeds

CRUD Operations

Create, **Read**, **U**pdate, and **D**elete

- **Read** Query for aggregations and the Design Document concept [CDB-DSD]
  - *Views* Querying and Aggregation w/ MapReduce
    - Restrict and aggregate documents from database with specific order
    - Indexing of documents for particular needs, and relationships
    - Computation is delivered as map-(re-)reduce program (written in JavaScript)

CRUD Operations

Create, **R**ead, **U**pdate, and **Delete**

- **Delete** Deletes database academic_graph (if existing) [CDB-GTS]

Bash
```
$ curl -X DELETE http://127.0.0.1:5984/academic_graph
```

JSON
```
{"ok": true}
```

HTTP **DELETE** method on database name to remove this database

CRUD Operations

Create, **R**ead, **U**pdate, and **Delete**

- **Delete** Deletes document by its id and (latest) revision number (if existing) [CDB-API]

Bash
```
$ curl -X DELETE http://127.0.0.1:5984/academic_graph/
                 <primary-key>?rev=<revision>
```

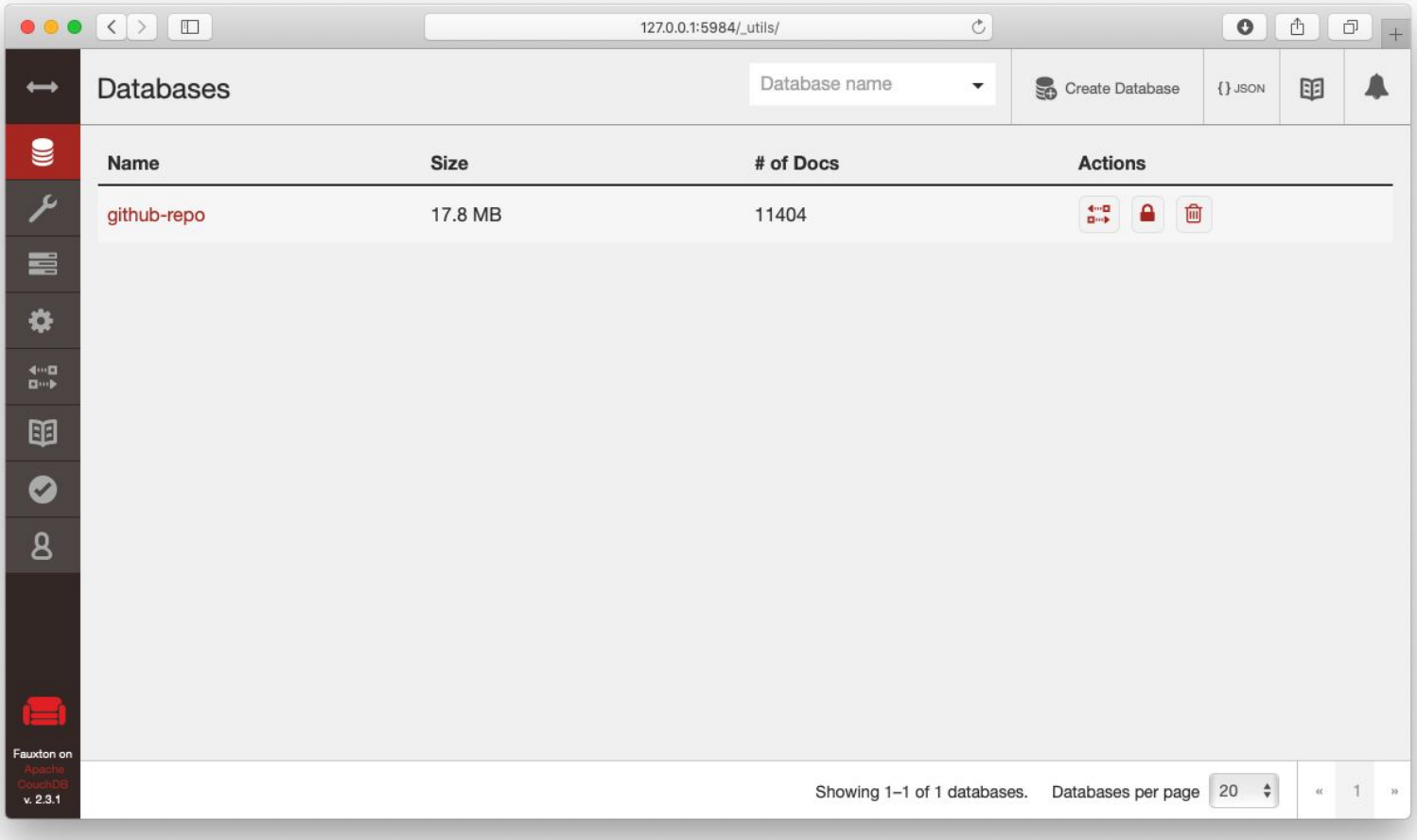
JSON
```
{"ok": true, "id"="primary-key", "rev"="<new-revision>"}
```

HTTP **DELETE** method on document id (`primary-key`) to identify document, and revision number to refer to version of document to delete

- Revision number must be latest revision number to resolve conflicts
  - CouchDB rejects deletion request if `revision` is not latest
    - Version conflicts handled via user-empowerment
  - May require to fetch current document (incl. current revision) first

CouchDB does not physically delete documents, instead a deletion adds a new revision `new-revision` marked as deleted. Retrieving previous version is possible, though.

CouchDB UI


127.0.0.1:5984/_utils/
Databases
Database name
Create Database
{} JSON
Name
Size
# of Docs
Actions
github-repo
17.8 MB
11404
Fauxton on Apache CouchDB v. 2.3.1
Showing 1–1 of 1 databases.
Databases per page
20
1

MapReduce

MapReduce (I)

Programming model and framework for robust processing large data collections by Google [DG-08]

- Computation is built for distributed, parallel execution
- Used for various computations, e.g., pattern-based search, inverted indexes
- Limited fit for iterative algorithm, e.g., Machine Learning tasks

A MapReduce program consists of two+ functions

- **map** Invoked over list of elements (original key-value pairs/single documents)
  - *purpose* **filtering or sorting**
  - each map takes a single (k_1, v_1) pair as input
  - each call returns (*emits*) a new key-value pair list $list(k_2, v_2)$
- **reduce** Retrieves a key along with a value list from map function
  - *purpose* **aggregation** (counting, summaries,…)
  - each reduce takes a single $(k_2, list(v_2))$ pair as input
  - each call returns a list of values $list(v_2)$
  - original Google MapReduce results in *n* result sets for *n* reducer
- re-reduce,… Implementation-specific extensions, such as running multiple reduces

MapReduce (II)

Example Original word count example [DG-08]

Pseudo

```
map(String key, String value): // key: document name, value: document contents
     for each word w in value:
          emit(w, "1");

reduce(String key, Iterator values): // key: a word, values: a list of counts
     int result = 0;
     for each v in values:
          result += ParseInt(v);
     emit(AsString(result));
```

MapReduce in mongoDB

Dedicated database command

mapReduce [MDB-RM]

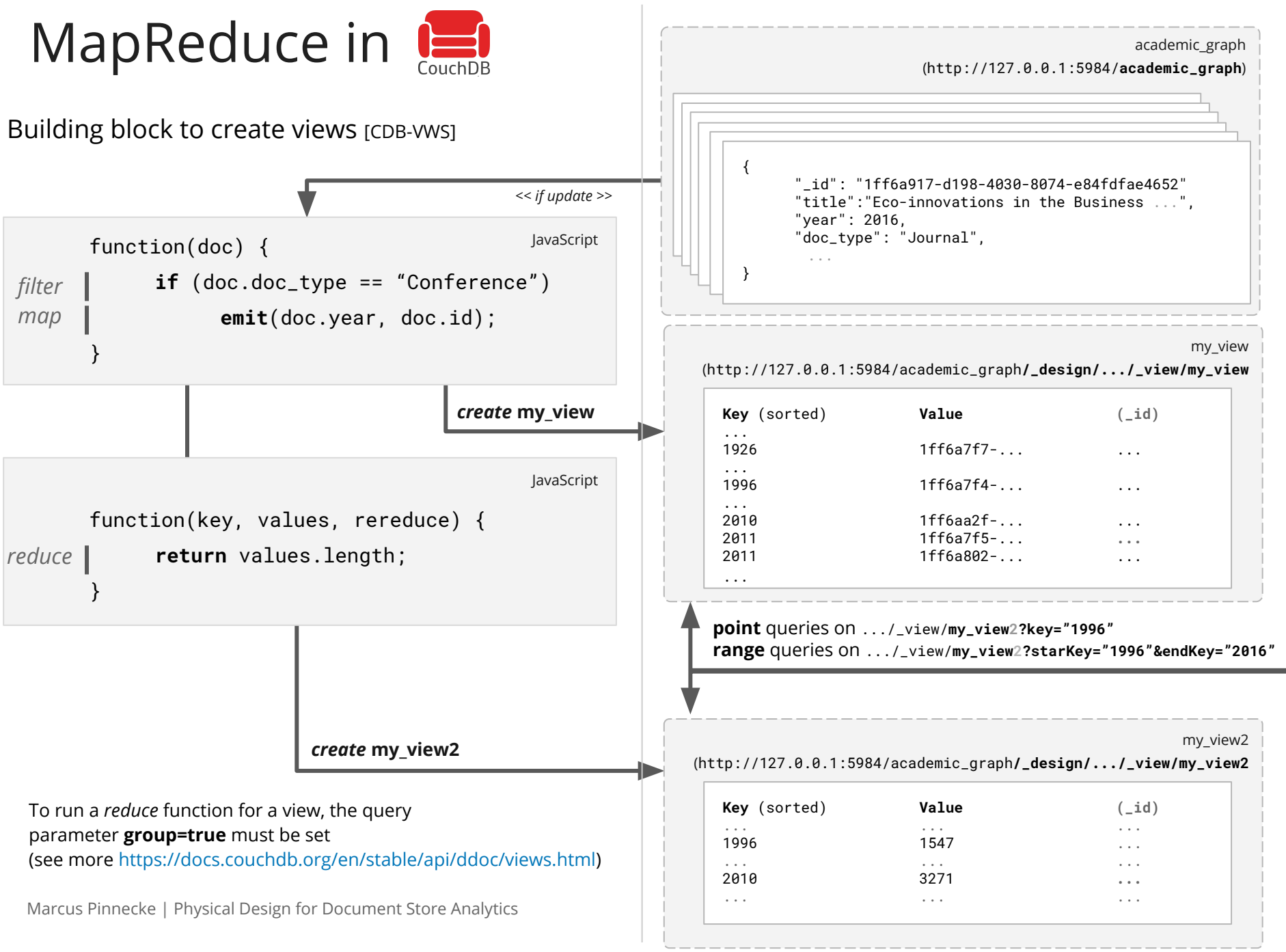
JavaScript

```
db.academicGraph.mapReduce(
    map:    function() {
                emit(this.year, this.id);
            },
    reduce: function(key, values) {
                return Array.count(values);
            },
    filter & output:
            {
                query: { doc_type: “Conference” },
                out: “papersPerYear”
            }
)
```

- Output is either intermediate or stored as a collection
  - *Incremental MapReduce* if stored as collection

MapReduce in CouchDB

Building block to create views [CDB-VWS]



To run a *reduce* function for a view, the query parameter **group=true** must be set (see more https://docs.couchdb.org/en/stable/api/ddoc/views.html)

Summary

Document Stores

Summary Document Stores

Document Stores Overview and Comparison

- Storage engine comparison - Append-Only vs Update-In-Place
- Different record formats and record organizations - JSON database vs BSON collections
- Query formulation, query language and database communication

CRUD (Create, Read, Update, Delete) Operations in mongoDB and CouchDB

- creation of databases, insertion of documents
- querying documents with filter operators, dot-notation, projection, sorting,...
- document identity (and for CouchDB revision management)
- aggregation query expression (and for CouchDB design documents)
- modification and deletion of databases and documents
- MapReduce as model and framework, usage and extensions in mongoDB vs CouchDB

OTTO VON GUERICKE UNIVERSITÄT MAGDEBURG

Document Stores

Storage Engine Overview

(System Land)

CouchDBs Storage Engine

Document Store Storage Organization

Append-Only Storage

- **Database modifications are logical *insert* operations**

  Insert create new document with new `_id`

  Update create new document with **old** `_id` and new revision number

  Delete create new document with **old** `_id` and tombstone marker

- **Any insert operation requires to update two files**

  Index-File serialized B^+-tree to support efficient range queries

  Database-File sequence of documents in order of insertions

  A (physical) document is identified by its `_id` and *never* modified once created

  pro less impact of faults on existing data, less random access in file

  con higher space requirements

  Concurrent reads during writes access last consistent database version by reading index file from its end towards its beginning.

Revision Control

Revision Control Version tracking of modifications (inserts, update, and deletes) to objects.

Revision Number Modification is manifested, a revision number is created and assigned

- Object version is identified by its revision number
- Set of revisions is (change) history
- Revisions can be compared, retrieved and merged

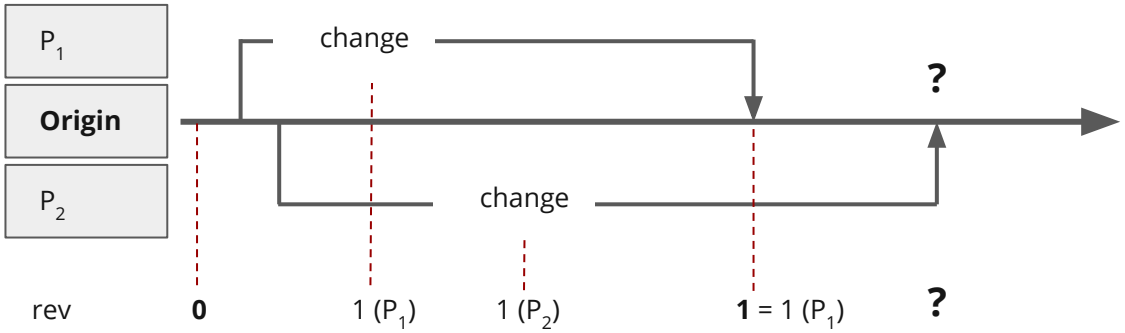
Examples

- *Software Development* Git, SVN,...
- *Databases* CouchDB,...

Revision Control (Conflict Handling)

Example

A has copies of document *D* stored (w/o sync) on two distinct places P_1, P_2.
A adds one information to $D(P_1)$ but not on $D(P_2)$, and vice-versa.
A performs a synchronization of *D* in P_1, P_2 such that $D(P_1) = D(P_2)$ shall hold.



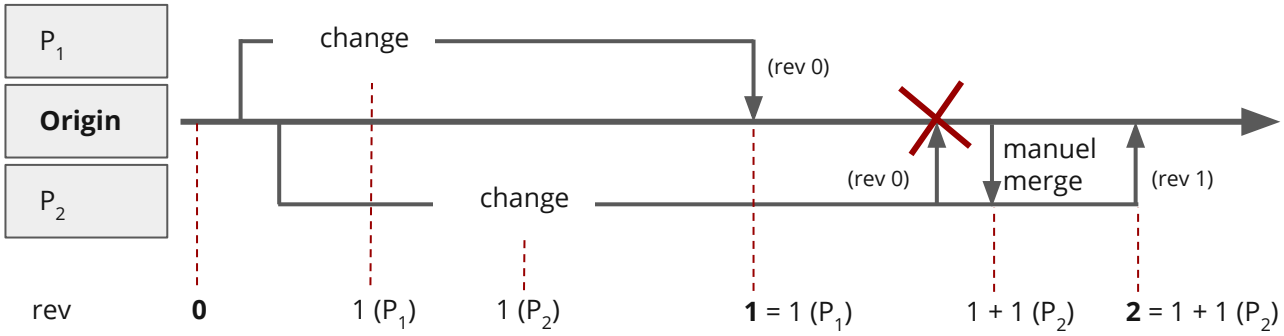
potential conflict: what happens to change at P_1 since P_2 operates on revision **0** -- especially if $1(P_2)$ is contradicting to $1(P_1)$?

Revision Control (Conflict Handling)

CouchDB [CDB-REV]

Example

A has copies of document *D* stored (w/o sync) on two distinct places P_1, P_2.
A add one information to $D(P_1)$ but not on $D(P_2)$, and vice-versa.
A performs a synchronization of *D* in P_1, P_2 such that $D(P_1) = D(P_2)$ shall hold.



"Conflict Avoidance" Solution in CouchDB is user-empowered MVCC

- When update is performed, current rev number must be specified
- If update rev number is outdated, update is rejected by CouchDB
  - "The one who saves first, wins"
  - Client may fetch latest revision first and perform merge himself

Exercise: *Alternatives to conflict avoidance? What happens in distributed case?*

MongoDBs Storage Engine

Document Store Storage Organization

Update-In-Place Storage

- **Database modifications are logical *insert* operations**

  Insert create new document with new `_id`

  Update modifies document but keeps `_id` (unless *upsert* is used)

  Delete set tombstone marker for `_id` (actual deletion is postponed)

  A (physical) document is identified by its `_id` and potentially modified (expect `_id` field)

  pro lower space requirements

  con more impact of faults on existing data, more random access in file

  Point-in-time snapshot of (in-memory view of) data to transactions that is written in intervals of 60sec to disk. Written snapshot is durable and acts as new checkpoint for recovery purposes. Old checkpoints get invalid (and freed) after successful write of snapshot as new checkpoint. Journaling (write-ahead transaction log) is optional .

WiredTiger

mongoDB WIREDTIGER

- **Traditional B$^+$-tree structure is used to organize key-value storage file**

  Row-Store keys and values are variable-length byte strings

  Column-Store keys are 64bit identifiers, values are fixed-/variable-length byte strings

  Log-Structured Merge Trees (LSM) implemented as tree of B$^+$-trees

  A (physical) document is potentially managed by different formats (e.g., sparse, wide table as column-store primary, and indexes as LSM tree)

  Compression is applied

  key prefix compression prefix is stored once per page (mem+disk, row-store only)

  dictionary compression identical values are stored once per page (mem+disk)

  huffman encoding compressing individual key/value items (mem+disk)

  block compression compresses blocks on backing file (disk)

  run-length encoding sequential, duplic. values stored only once (mem+disk, column-store only)

Physical Record Organization

- or -

Organizing Semi-Structured Data with Bits and Bytes

Physical Record Organization (I)

Why should you care about different physical formats in the first place?

Physical Record Organization (II)

- **Required** Physical format is needed to effectively work with JSON-like data (obviously)
  - Even if “Plain-Text JSON” is used, you have *one* possible implementation of the concept
- **Diversity** Different requirements, and different purposes call for alternatives
  - *Fast Parsability* Binary encoding rather than plain text (BSON, UBJSON, CARBON,...)
  - *Understandability* Human-readability independent of encoding (JSON, UBJSON, ...)
  - *Accessibility* Low entry barrier to use format across systems (JSON, UBJSON,...)
  - *Expressibility* Support of non-standard data types, e.g., spatial data (BSON,...)
  - *Simplicity* Restriction to standard data types satisfying RFC 8259 (JSON, UBJSON,...)
  - *Indexability* Specialized format to be integrated into existing system (JSONb, CARBON, ...)
  - *Compactability* Low (runtime, persistent) memory footprint (UBJSON, CARBON, ...)
  - *Cache Efficiency* Processor data-prefetcher optimized layout (CARBON, ...)
- **No “One-Size-Fits-All”** No single format to “rule them all” due to trade-off decisions (e.g., *expressibility* vs *simplicity*), or contradicting optimization (cf., *row-wise* vs *columnar* layout)

Physical Record Organization (III)

Formats suitable for database purpose (object representation or persistence**)**

- Plain-Text JSON **JSON**
- Universal Binary JSON **UBJSON**
- mongoDBs Binary JSON **BSON**
- Postgres' Binary JSON **JSONb**
- NG5s Columnar Binary JSON **CARBON**

Formats for other purpose (network communication, data exchange, or general purpose**)**

- Google ProtocolBuffers, CBOR, MessagePack, and others

Plain-Text JSON (I)

An UTF-8 encoded plain-text string
satisfying the syntax in RFC 8259.

Who By Internet Engineering Task Force (IETF); first appeared in 1996

Goal Portable representation of structured data for data interchange, strictly implementing RFC 8259

What A flat-file, lightweight, text-based, human-readable, and language-independent format (extension `.json`)

Use Favored form for network communication & REST-based services, CouchDBs records

Implementers Various libraries by different vendors

www.json.org

Plain-Text JSON (II)

paper1.json

```
{ "title": "Structural defects in GaN", "authors": [ { "name": "S. Ruvimov", \
"org": "Div. of Mater. Sci (...)" }, { "name": "Z. Liliental-Weber" } ],     \
"references": [ "07d52a00-109f(...)", "48f2de10-2c83(...)",                  \
"6d1efe54-c7aa(...)", "c2950b99-d734(...)", "ccab2fc4-276d(...)",            \
"df0e1313-9b65(...)" ] }
```

paper2.json

```
{ "title": "A decision support tool", "authors": [ { "name": "Charles White" } ] }
```

Universal Binary JSON - UBJSON (I)

A lightweight binary-encoded human-readable JSON format fully compatible to JSON Spec of March 2014 (RFC 7159).

Riyad Kalla
Director, Global Consumer Credit at PayPal

Who By Riyad Kalla; rooted back to Sep 2011 (or earlier) with initial library commit

Goal Strict compatibility to JSON spec to match native type support in all major programming languages, simplicity of specification and low adaption barrier for developers, and fast parsing and low memory footprint.

What A flat-file, lightweight, binary-encoded, type-marker based, human-readable, and language-independent format (extension `.ubj`)

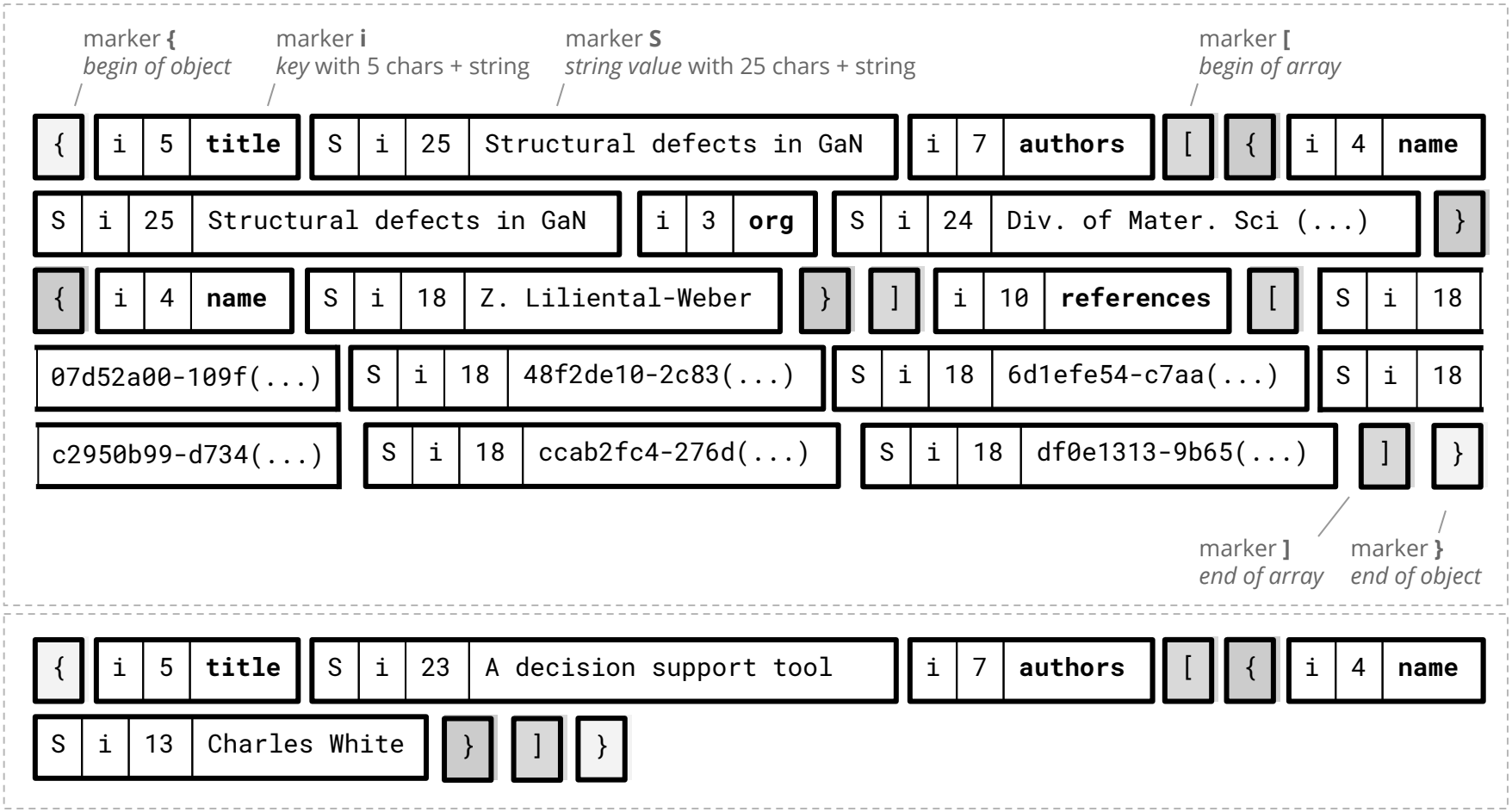
```
[type, 1-byte char]([integer numeric length])([data])
```

Type Marker Data Format of UBJSON

Implementers Libraries for ASM.JS, C/C++, D, Go, Java, JavaScript, MATLAB, .NET, Node.js, PHP, Python, Qt, and Swift by various vendors

www.ubjson.org

Universal Binary JSON - UBJSON (II)



Binary JSON - BSON (I)

An expressive binary-encoded JSON format partially compatible to JSON Spec to store JSON-like records.

Who By 10gen Inc. (now MongoDB Inc.); before 1st release of MongoDB in 2009

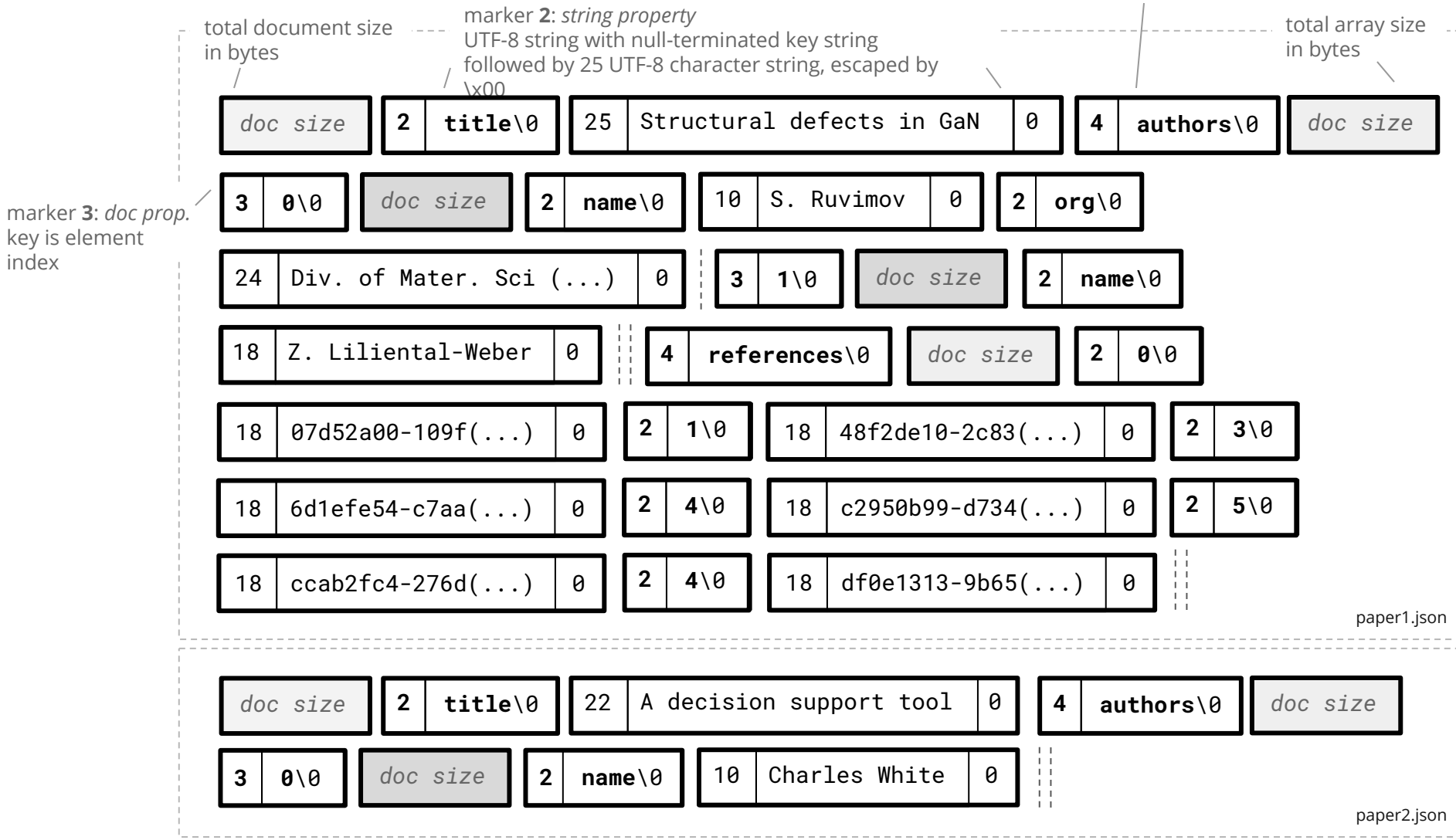
Goal Low memory footprint for metadata and small binary size to optimize for network communication, easy traversable to support data access in MongoDB, fast encoding to and decoding from BSON for data exchange.

What A flat-file, non-JSON-standard, data-type rich, lightweight, binary-encoded, and language-independent format for communication with and processing in MongoDB (extension `.bson`). An array *a* is an object *o* where *i*-th element *e* in *a* is property (i, *e*) in *o*.

Implementers C library (libson) used in MongoDB, additional bindings for .NET, C++, D, Dart, Delphi, Exlixir, Erlang, Factor, Fantom, Go, Haskell, Java, Lisp, Lua, Node.js, OCaml, Perl, PHP, Prolog, Python, Ruby, Rust, Scala, Smalltalk, SML, and Swift.

www.bsonspec.org

Binary JSON - BSON (II)



Columnar Binary JSON - CARBON (I)

A traversal-optimized binary format partially compatible to RFC 8259 to store read-mostly JSON-like record collections.

Marcus Pinnecke
Research associate at University of Magdeburg

Who By Marcus Pinnecke; rooted back to Nov 2018; still in research and dev

Goal Main-memory optimized data layout for fast SQL/JSON filter expression evaluations, compatibility to majority of JSON files, fast traversals in huge “cold-data” document database partitions (named *archives*), low memory footprint for archives in memory and disk, and wire-speed loading of archives parts into memory.

What A non flat-file, non-JSON-standard, binary-encoded, type-marker based, variable-structured, index built-in, metadata rich, language-independent read-only JSON collection format with built-in object identification, and smart compression (extension `.carbon`). Carbon file consists of a (compressed) string table kept on disk, and a memory resident record table that is instantly loaded.

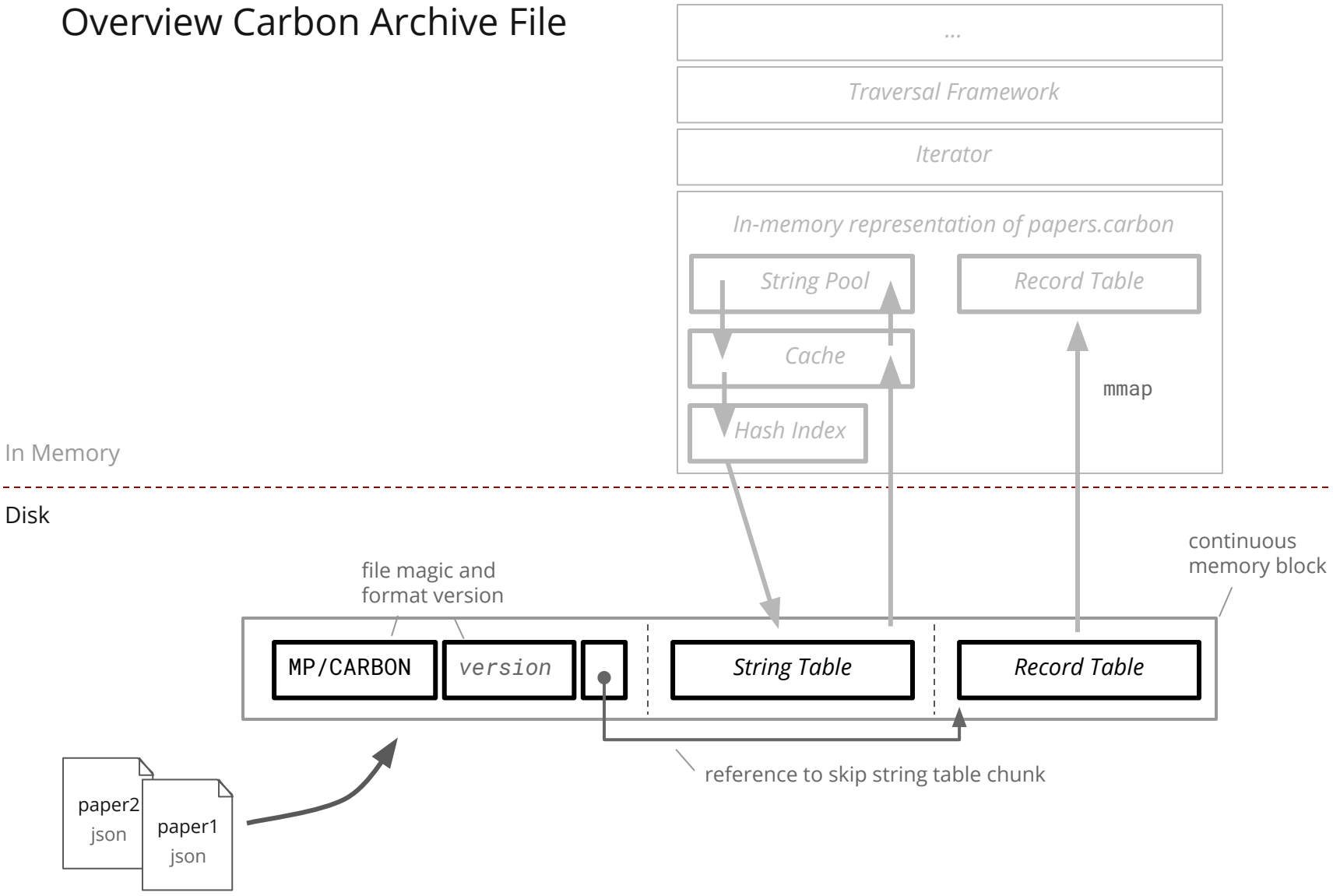
Elements must have same (nullable) type inside arrays.

Implementers C library (libcarbon) with in storage engine NG5 (*engine 5*).

www.carbonspec.org and www.github.com/protolabs/libcarbon

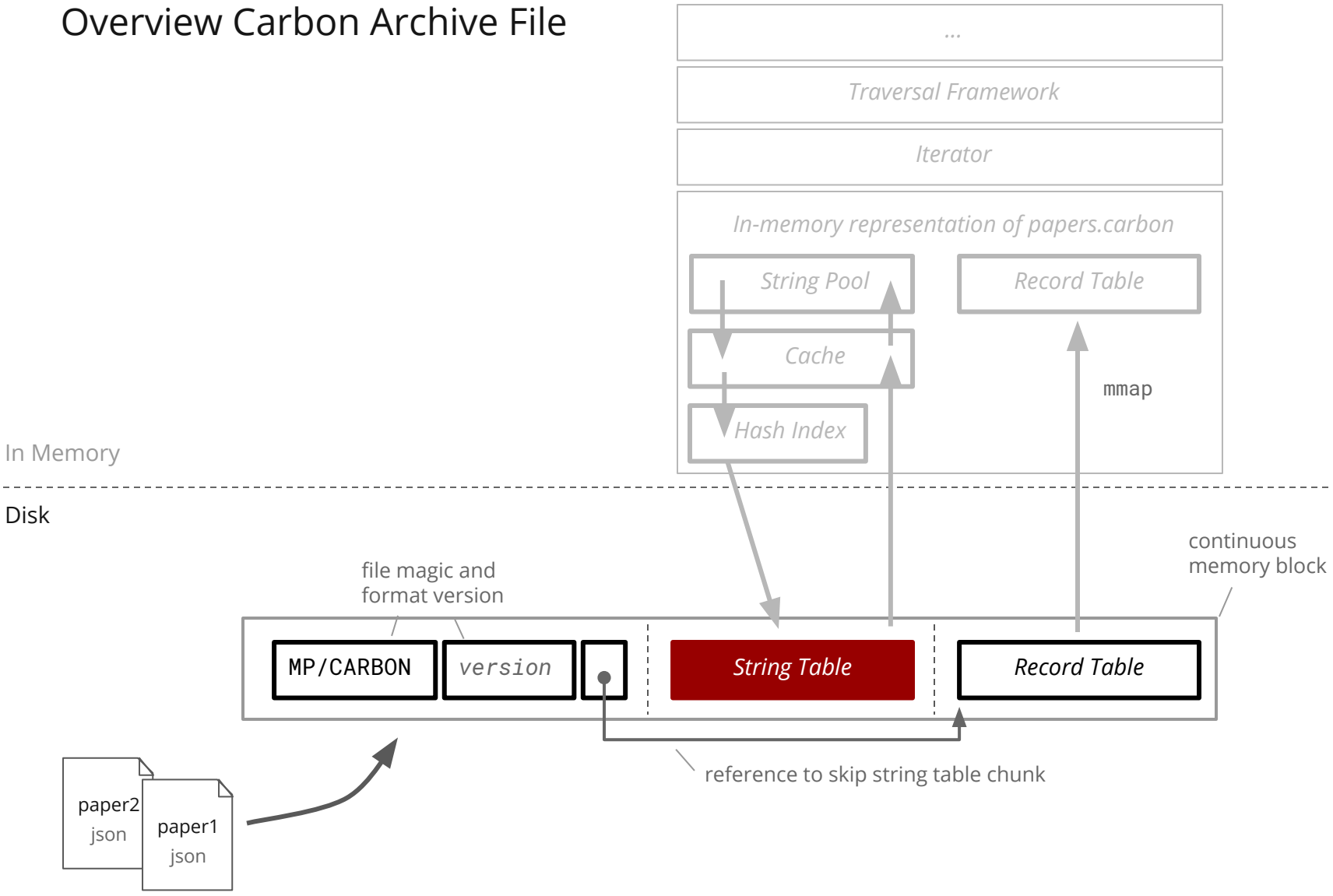
Columnar Binary JSON - CARBON (II)

Overview Carbon Archive File



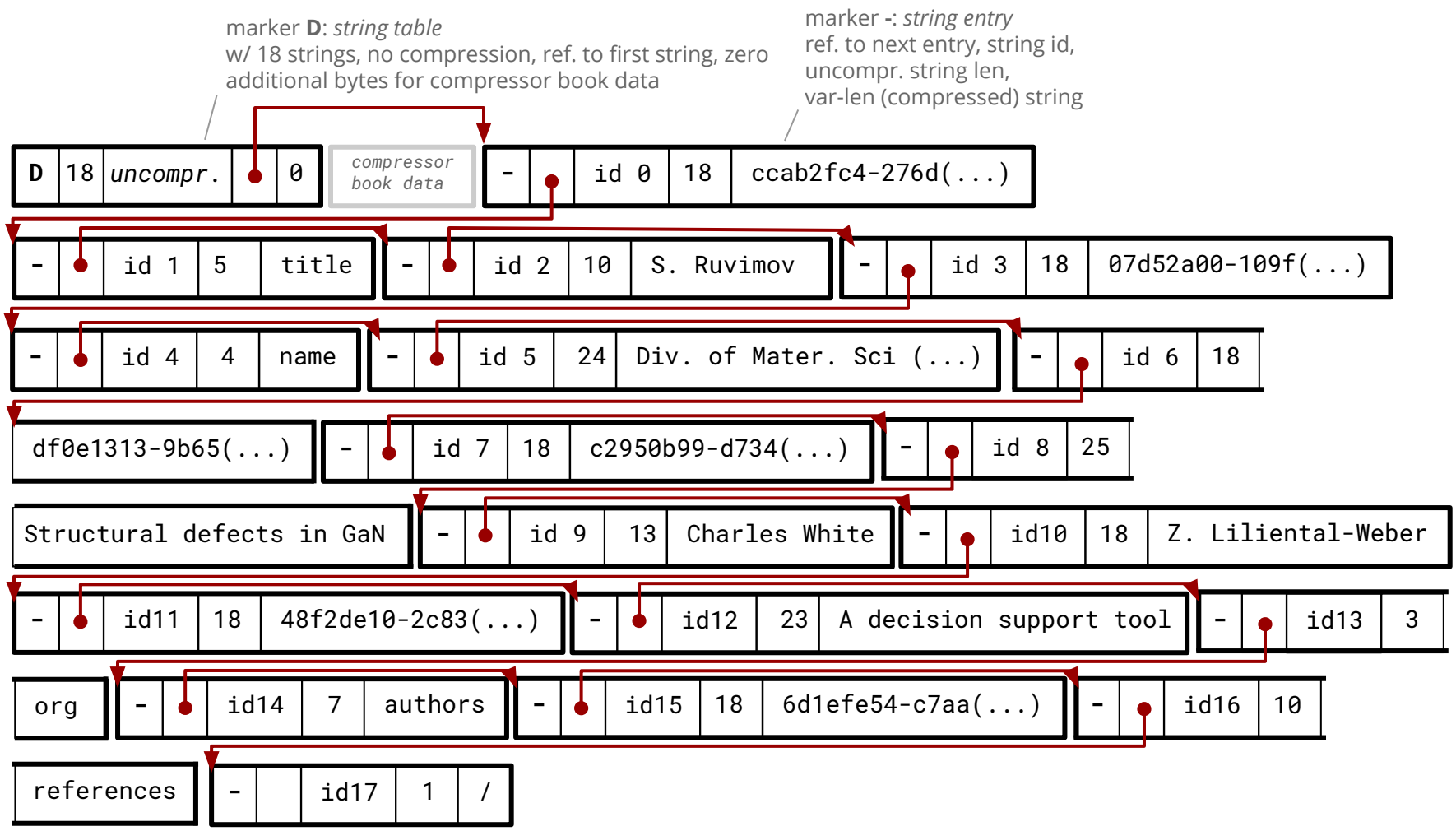
Columnar Binary JSON - CARBON (II)

Overview Carbon Archive File



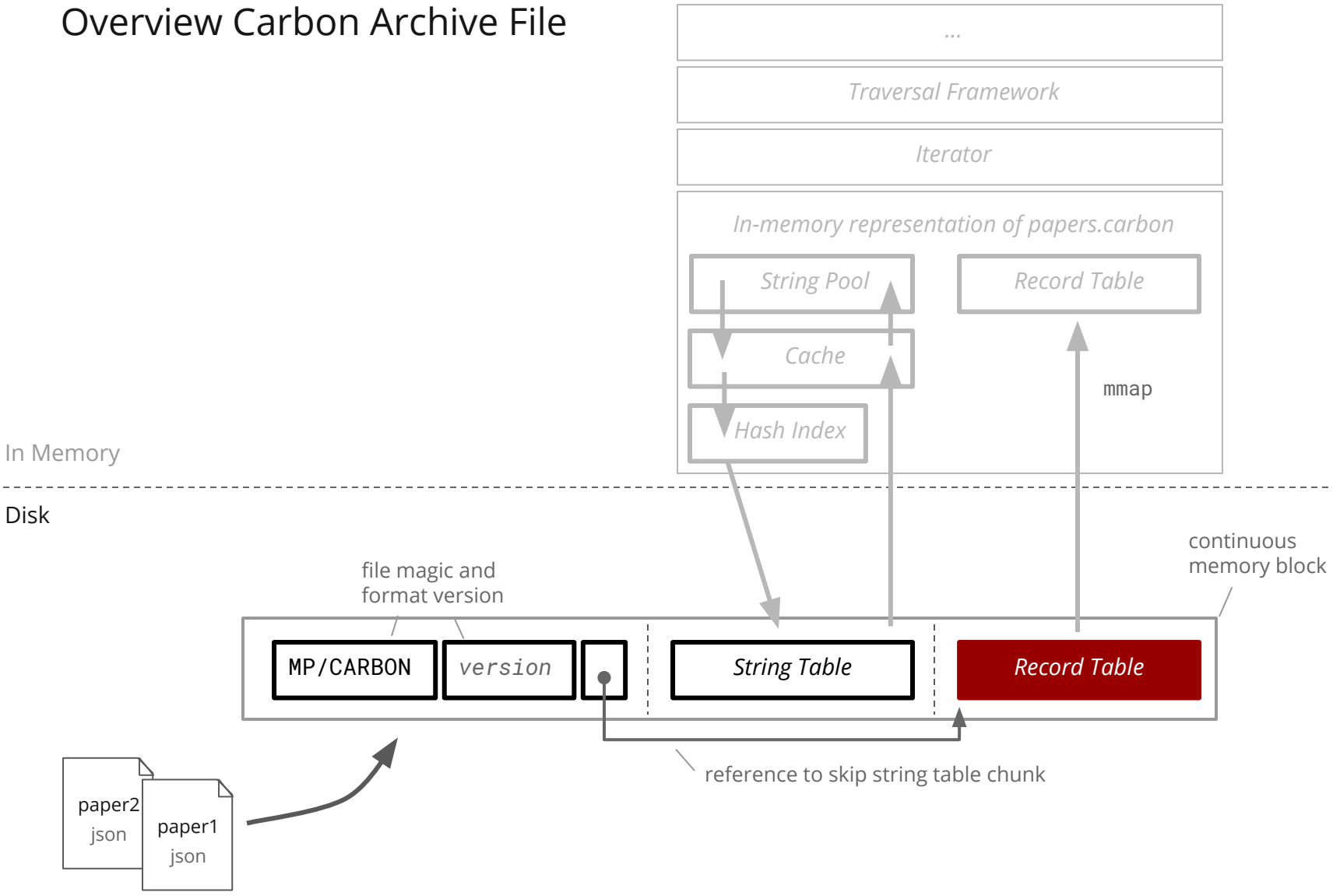
Columnar Binary JSON - CARBON (III)

String Table

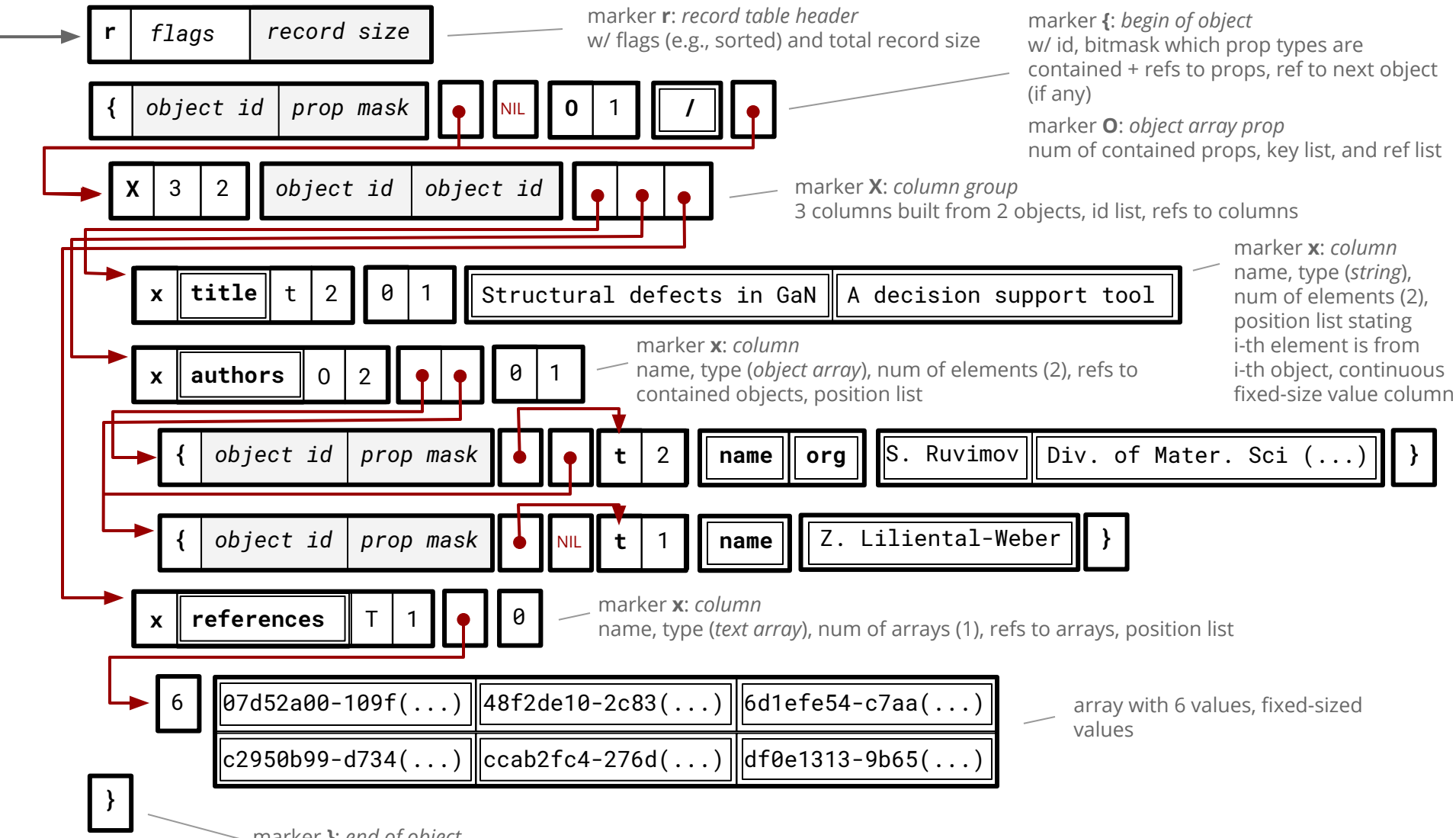


Columnar Binary JSON - CARBON (IV)

Overview Carbon Archive File



Columnar Binary JSON - CARBON (V)



s — Fixed-length string id for string **s** (i.e., reference into string table).
Variable-length string **s** given in Figure for ease of understanding, only.

Columnar Binary JSON - CARBON (VI)

CARBON enables efficient traversal in schema out-of-the-box, and access to continuous (fixed-sized) value columns across documents sharing same attribute (key + type) while at same time is competitive in total binary size.

For n documents stored in a database (collection), with m keys in each document:

	CARBON	**Flat-files**
• schema traversal	$\mathcal{O}(m)$	$\mathcal{O}(nm)$
• value access across docs for fixed key	$\mathcal{O}(1)$	$\mathcal{O}(n)$

Summary

Storage Engine Overview

Summary Storage Engine Overview

Insights into one Append-Only and one Update-In-Place storage engine

- Database modifications and what happens underneath
- Document identity (document id), revision control and its application in CouchDB
- Multi-version management in CouchDB and MongoDB
- Discussion of pros and cons
- Insights into key properties of WiredTiger (MongoDBs storage engine)

Physical Record Organization

- Overview on representation formats for JSON-like records
- Key properties and example for Plain-Text JSON, UBJSON, BSON & CARBON
- CARBON archive file overview, complexity comparisons

OTTO VON GUERICKE UNIVERSITÄT MAGDEBURG

JSON Documents in Relational Systems

JSON Support in Relational Database Systems

(...)

SQL/JSON Standard

JSON in SQL:2016 Standard

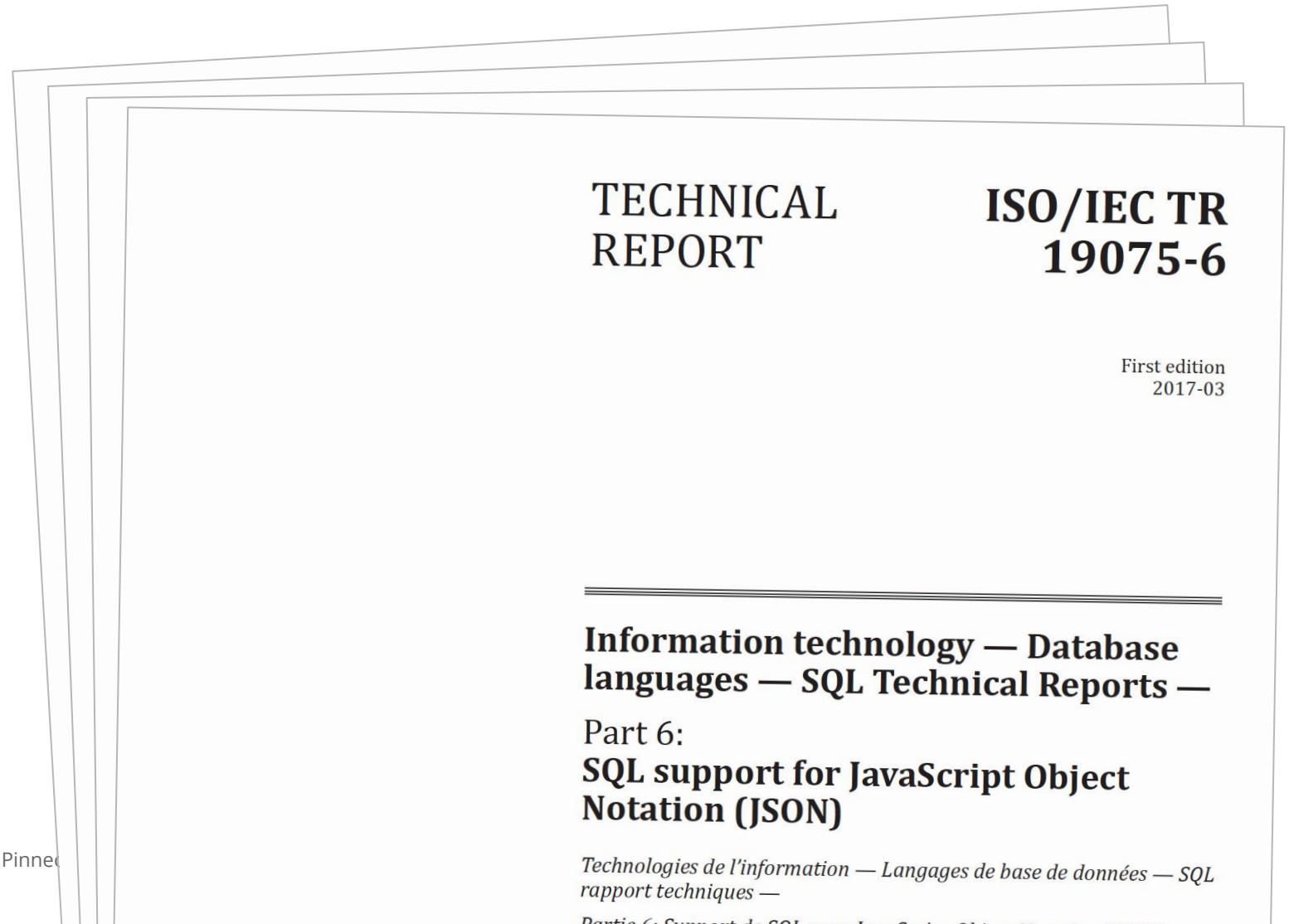
SQL Standard

SQL as *the* standard to query structured data (e.g., in relational database systems)

- Initiated 1974 by Chamberlin and Boyce (IBM) [SEQ-UEL]
- Bases and extends concepts of relational algebra and tuple calculus
- Consists of
  - *clauses* like **`SELECT`**, **`FROM`**, **`WHERE`**, **`UPDATE`**, ...
  - *expressions* returning scalars or tables
  - *predicates* returning **`true`**/**`false`**/**`null`**
  - *statements* data querying, definition, manipulation and control
- Latest standard (**SQL:2016**) **adds JSON support** to the language

SQL:2016 Support for JSON

(roughly 90 pages of content)



TECHNICAL REPORT

ISO/IEC TR 19075-6

First edition
2017-03

Information technology — Database languages — SQL Technical Reports —

Part 6:
SQL support for JavaScript Object Notation (JSON)

Technologies de l'information — Langages de base de données — SQL rapport techniques —

SQL:2016 SQL/JSON (I)

New feature set in SQL to support JSON [ISO-SQL, SQL-16]

- JSON as *string* type rather than a dedicated native type (like XML)
- Standard is not fully implemented in commercial systems or vendor-specific adapted:
  - Validation Function
  - Construction Functions
  - Query Functions
  - SQL/JSON Path Language

SQL:2016 SQL/JSON (II)

New feature: Validation Function [ISO-SQL, SQL-16]

```
<expr> is [not] json [value | array | object | scalar ]
```

New predicate `is json` to check if value is a well formed JSON string

SQL:2016

```
is json '{ "authors":[ { "name":"Charles White" } ] }'
```

SQL:2016 SQL/JSON (III)

New feature: Construction Functions [ISO-SQL, SQL-16]

```
json_object([key] <expr> value <expression> [,...])
json_objectagg([key] <expr> value <expression>)
```

Create a new JSON object string from key-/value pairs (of a group)

SQL:2016

```
json_object(key 'last-name'  value 'Pinnecke',
            key 'first-name' value 'Marcus')
```

JSON

```
{ "last-name": "Pinnecke",
  "first-name": "Marcus" }
```

SQL:2016

```
SELECT group-col, json_object(key-col value value-col)
FROM ...
GROUP BY group-col
```

Table Print

```
+----+--------------------------+
| g1 |  {"k1": "v1", "k2": "v2"} |
| g2 |  {"k3": "v3"}             |
+----+--------------------------+
```

SQL:2016 SQL/JSON (IV)

New feature: Construction Functions [ISO-SQL, SQL-16]

```
json_array([<expr>][,...])
json_array(<query>)
json_arrayagg(<expr> [order by ...])
```

Create a new JSON array string from values, from a query result, or from values of a group.

SQL:2016

```
json_array(1,2,3,4)
```

SQL:2016

```
json_array(SELECT col FROM ...)
```

SQL:2016

```
SELECT json_arrayagg(col ORDER BY ...)
FROM ...
GROUP BY ...
```

JSON

```
[1,2,3,4]
```

SQL:2016 SQL/JSON (V)

New feature: Query Functions [ISO-SQL, SQL-16]

`json_exists(<json-col>, <path>)`

Tests if specific path `<path>` exists in JSON string for each row in column `<json-col>`.
Results `true`, `false`, or `unknown`, can be placed in `WHERE` clause

SQL:2016

```
...
WHERE json_exists(docs, '$.authors')
```

SQL:2016 SQL/JSON (VI)

New feature: Query Functions [ISO-SQL, SQL-16]

```
json_value(<json>, <path> [returning <type>])
```

Gets a scalar value (no object, no array) from JSON string `<json>` given JSON Path `<path>`. Returns a SQL datum, optionally type-cased to `<type>` (default is string). Fails for multiple hits.

SQL:2016

```
json_value('{
               "authors":[
                    { "name": "S. Ruvimov",
                      "org":  "Div. of Mater. Sci (...)" },
                    { "name":"Z. Liliental-Weber" }
                ]
          }',
          '$.authors[1].name'
        )
```

Table Print

```
+--------------------+
| Z. Liliental-Weber |
+--------------------+
```

SQL:2016 SQL/JSON (VII)

New feature: Query Functions [ISO-SQL, SQL-16]

json_query(<json>, <path> [**with** [**conditional** | **unconditional**] [**array**] **wrapper**])

Like **json_value** but extracts any value (incl. arrays and objects) from JSON string <json>. Returns a JSON string. Special treatment for multiple hits: fail, add if needed, or force force surrounding with array braces [ ]

SQL:2016

```
json_query('{
               "authors":[
                    { "name": "S. Ruvimov",
                      "org":  "Div. of Mater. Sci (...)" },
                    { "name":"Z. Liiental-Weber" }
                ]
            }',
            '$.authors[*].name'
            with wrapper
           )
```

JSON

```
[ "S. Ruvimov",
  "Z. Liiental-Weber" ]
```

SQL:2016 SQL/JSON (VIII)

New feature: Query Functions [ISO-SQL, SQL-16]

`json_table(<json-col>, <path> columns ...)`

Converts JSON objects that match `<path>` within a JSON string column `<json-col>` to rows in a table. Per-row column values are (potentially) extracted with a JSON path language query the corresponding object.

Table Print

```
+-----------------------------------+
| docs                              |
+-----------------------------------+
| { "x": 1, "y": { "m": 2, "n": 3} } |
| { "a": 4 }                        |
| { "x": 5, "y": { "m": 6 } }       |
+-----------------------------------+
```

SQL:2016

```
SELECT t.*
FROM json_table(
      docs, '$.x',
      columns (a NUMERIC path '$.y.m',
            b VARCHAR(100) path '$.y.n')
               ) t
```

Table Print

```
+-------+
| a | b |
+-------+
| 2 | 3 |
| 6 |   |
+-------+
```

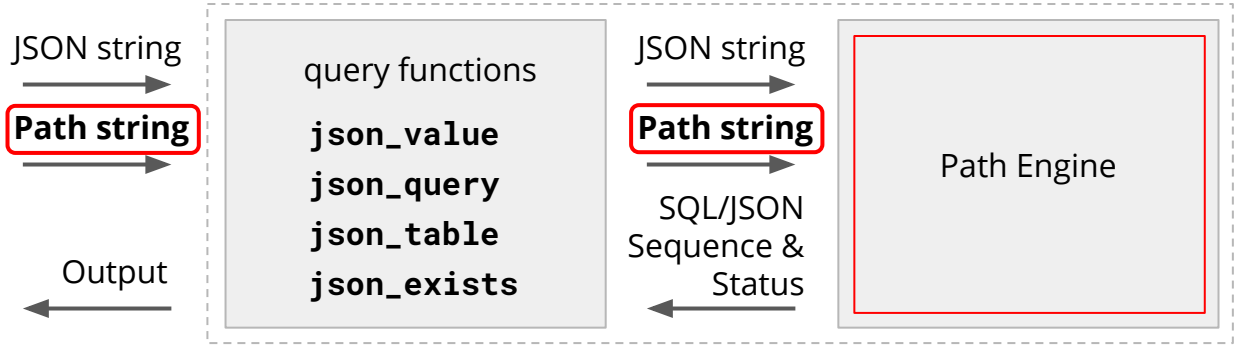
SQL:2016 SQL/JSON Path Language

SQL:2016

```
SELECT t.*
FROM json_table(
     docs, '$.x'
     columns (a NUMERIC path '$.y.m',
           b VARCHAR(100) path '$.y.n')
             ) t
```

SQL/JSON Path Language (I)

```
,
 NUMERIC path '$.y.m',
RCHAR(100) path '$.y.n')
t
```



Architecture of SQL/JSON Path Language

(based on [ISO-SQL] p. 55)

SQL/JSON Path Language (II)

SQL/JSON Path Language is a query language embedded in SQL [ISO-SQL]

SQL/JSON Path Language

```
'lax $.authors.name ? (@ starts with "Pinn")'
```

- Used in SQL/JSON query functions (**`json_value`**, **`json_query`**, **`json_table`**, **`json_exists`**)
- Function/predicate semantic based on SQL semantics
  - Especially, whole path expression must be SQL quoted (single quote `'<path-str>'`)

SQL/JSON Path Language (III)

SQL/JSON Path Language is a query language embedded in SQL [ISO-SQL]

SQL/JSON Path Language

```
'lax $.authors.name ? (@ starts with "Pinn")'
```

- JavaScript-inspired (e.g., `.` (dot) member access, `[ ]` array access, 0-indexed arrays,...)
  - Query language is case-sensitive (in contrast to SQL itself)
  - Variable names start with `$` (dollar), or as key-name after `.` (period)
  - String literals are enclosed with double quotes (`"<str>"`)
  - Path evaluation with mode
    - `lax` arrays of size 1 $\asymp$ to single element
      arrays are unnested automatically
      if key not exists (or other structural error), empty result is returned
    - `strict` arrays of size 1 $\not\asymp$ to single element
      arrays are not unnested automatically
      if key not exists (or other structural error), error condition is returned

$\asymp$ ... equivalent

Data Model

SQL/JSON Path Language Data Model (I)

- JSON with querying facilities in SQL as “embedded language” with own data model
- Several terms are used to distinguish between SQL, JSON, and SQL/JSON Path Langauge
  - “JSON” refers to any representation that is a JSON document [RFC7159]
  - “SQL/JSON” refers to JSON construct within SQL
- Well-defined parsing/serialization between JSON and SQL/JSON

SQL/JSON Path Language Data Model (II)

Terms in SQL/JSON Path Language

SQL/JSON		**JSON**
• SQL/JSON array, object, member, null	↦	array, object, member, literal `null`
• SQL `True`, `False`	↦	literal `true`, literal `false`
• (non-*null*) number	↦	number
• (non-*null*) character string	↦	string
• SQL datetime	↦	*(none)*
• SQL/JSON item	↦	*(none)*
• SQL/JSON sequence	↦	*(none)*

SQL/JSON Path Language Data Model (III)

SQL/JSON item (Def)

Recursively defined by

1. **SQL/JSON scalar** — non-null value of any SQL type (character string set, numeric, boolean, datetime)
2. **SQL/JSON null** — a value distinct from any SQL type value and SQL null value (i.e., a dedicated *null* value by its own)
3. **SQL/JSON array** — (potentially empty) ordered list of SQL/items (called **SQL/JSON elements** of SQL/JSON array)
4. **SQL/JSON object** — (potentially empty) unordered collection of **SQL/JSON members** (SQL/JSON member is *key-value* pair where *key* is character string and *value* is SQL/JSON item (called **bound value**))

SQL/JSON Path Language Data Model (IV)

SQL/JSON sequence (Def)

unnested, potentially empty ordered list of SQL/JSON items

Language Syntax

SQL/JSON Path Language Syntax (I)

SQL/JSON Path Language Syntax [ISO-SQL]

- Literals
  - `"string"`
  - `4.2e23`
  - **`true`**
  - **`false`**
  - **`null`**
- Variables
  - **`$`** — context item
  - **`$`**`name` — passed from SQL to expression
  - `@` — value of current item in filter
- Parentheses
  - `($a + $b)*$c`
- Accessors
  - `$.<name>`, `$."<name>"` — property with key <name>
  - `$."$<var>"` — property with value of variable <var>
  - `$.*` — wildcard property access
  - `$[1, 2, 4 to 7]` — array element accessor
  - `$[*]` — wildcard array element access

SQL/JSON Path Language Syntax (II)

SQL/JSON Path Language Syntax [ISO-SQL]

- Filter `$? (@.n_citation > 42)`
- Boolean `&&` `||` `!`
- Comparison `==` `!=` `<>` `<` `<=` `>` `>=`

SQL/JSON Path Language Syntax (III)

SQL/JSON Path Language Syntax [ISO-SQL]

- Predicates

```
exists ($)
($a == $b) is unknown
$ like_regex "colou?r"
$ starts with $a
```

- Arithmetics

```
+
-
*
/
%
```

SQL/JSON Path Language Syntax (IV)

SQL/JSON Path Language Syntax [ISO-SQL]

- Item functions
  - `$.type()`
  - `$.size()`
  - `$.double()`
  - `$.ceiling()`
  - `$.floor()`
  - `$.abs()`
  - `$.datetime()`
  - `$.kevalue()`

Variables

SQL/JSON Path Language Variables

Two types of variables

- **Context variable** `$` Path language always start with $
  Refers to the passed JSON string

SQL:2016

```
json_value('{ "num": 42 }', '$.num' )
```

- **Named variables** `$<name>` Additional variable given to path engine via **`passing`** clause

SQL:2016

```
json_value(T.docs, '$.values[$K]' passing T.pos as K )
```

Member Access

SQL/JSON Path Language Member Access (I)

Member access via `.` (dot) evaluation semantics

1. **Operator evaluation**
   Results in sequence of SQL/JSON items
2. **(a) In *strict* mode**
   Each SQL/JSON item in sequence must be object having specified key.
   If key does not exist, an error is returned.
   (b) In *lax* mode
   Each SQL/JSON array in sequence is unwrapped (unnested) one level as intermediate step.
3. **Iterate over values**
   Each SQL/JSON item is bound to value of specified key

SQL/JSON Path Language Member Access (II)

Example (*lax* mode): Access a property that does not exist for all array entries

JSON

```
{    "authors": [ {   "name": "S. Ruvimov",
                      "org":  "Div. of Mater. Sci (...)" },
                  { "name":"Z. Liliental-Weber" }
                ]
}
```

SQL/JSON Path Language

```
lax $
```

JSON

```
{    "authors": [ {   "name": "S. Ruvimov",
                      "org":  "Div. of Mater. Sci (...)" },
                  { "name":"Z. Liliental-Weber" }
                ]
}
```

SQL/JSON Path Language Member Access (III)

Example (*lax* mode): Access a property that does not exist for all array entries

JSON

```
{    "authors": [ {   "name": "S. Ruvimov",
                      "org":  "Div. of Mater. Sci (...)" },
                    { "name":"Z. Liliental-Weber" }
                  ]
}
```

SQL/JSON Path Language

```
lax $.authors
```

Intermediate *unwrap*

```
        {   "name": "S. Ruvimov",
            "org":  "Div. of Mater. Sci (...)" }
```

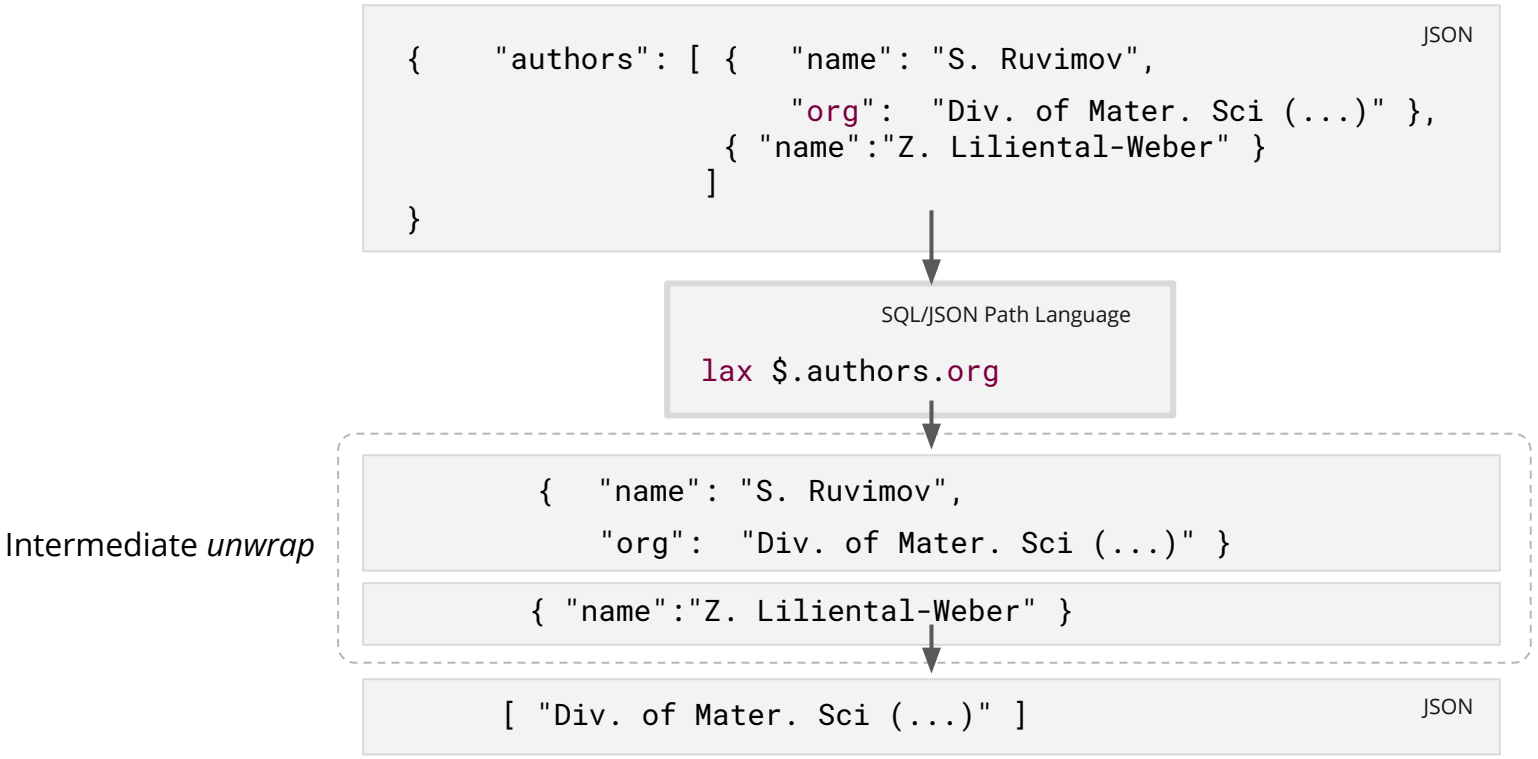
```
        { "name":"Z. Liliental-Weber" }
```

JSON

```
      [ {   "name": "S. Ruvimov",
            "org":  "Div. of Mater. Sci (...)" },
        { "name":"Z. Liliental-Weber" }
      ]
```

SQL/JSON Path Language Member Access (IV)

Example (*lax* mode): Access a property that does not exist for all array entries



SQL/JSON Path Language Member Access (V)

Example (*strict* mode): Access a property that does not exist for all array entries

JSON

```
{    "authors": [ {   "name": "S. Ruvimov",
                      "org":  "Div. of Mater. Sci (...)" },
                  { "name":"Z. Liliental-Weber" }
                ]
}
```

SQL/JSON Path Language

```
strict $
```

JSON

```
{    "authors": [ {   "name": "S. Ruvimov",
                      "org":  "Div. of Mater. Sci (...)" },
                  { "name":"Z. Liliental-Weber" }
                ]
}
```

SQL/JSON Path Language Member Access (VI)

Example (*strict* mode): Access a property that does not exist for all array entries

JSON

```
{    "authors": [ {   "name": "S. Ruvimov",
                      "org":  "Div. of Mater. Sci (...)" },
                    { "name":"Z. Liliental-Weber" }
                  ]
}
```

SQL/JSON Path Language

```
strict $.authors
```

JSON

```
[ {   "name": "S. Ruvimov",
      "org":  "Div. of Mater. Sci (...)" },
  { "name":"Z. Liliental-Weber" }
]
```

SQL/JSON Path Language Member Access (VII)

Example (*strict* mode): Access a property that does not exist for all array entries

JSON

```
{    "authors": [ {   "name": "S. Ruvimov",
                      "org":  "Div. of Mater. Sci (...)" },
                  { "name":"Z. Liliental-Weber" }
                 ]
}
```

SQL/JSON Path Language

```
strict $.authors[*]
```

Intermediate *unwrap*

```
        {   "name": "S. Ruvimov",
            "org":  "Div. of Mater. Sci (...)" }
```

```
        { "name":"Z. Liliental-Weber" }
```

JSON

```
      [ {   "name": "S. Ruvimov",
            "org":  "Div. of Mater. Sci (...)" },
        { "name":"Z. Liliental-Weber" }
      ]
```

SQL/JSON Path Language Member Access (VIII)

Example (*strict* mode): Access a property that does not exist for all array entries

JSON

```
{    "authors": [ {   "name": "S. Ruvimov",
                      "org":  "Div. of Mater. Sci (...)" },
                  { "name":"Z. Liliental-Weber" }
                ]
}
```

SQL/JSON Path Language

```
strict $.authors[*].org
```

Intermediate *unwrap*

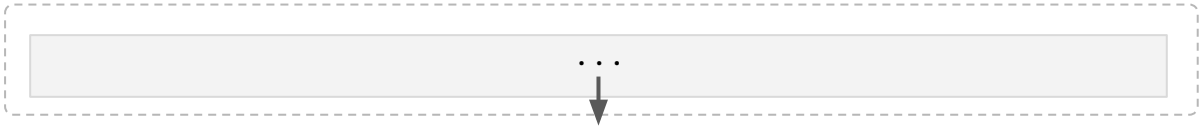
```
{   "name": "S. Ruvimov",
    "org":  "Div. of Mater. Sci (...)" }
{ "name":"Z. Liliental-Weber" }
```

Error is returned

(2nd object does not have property with key `org`)

SQL/JSON Path Language Member Access (IX)

Example (*strict* mode): Access a property that does not exist for all array entries



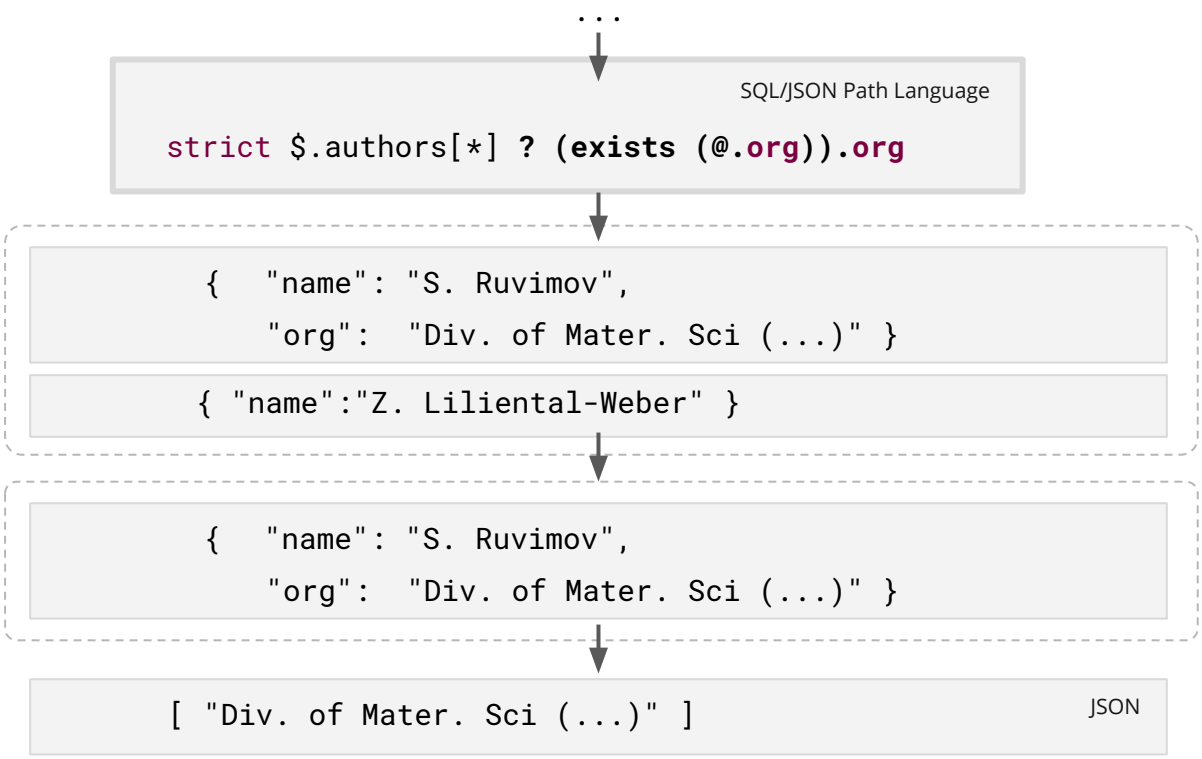
Error is returned

(2nd object does not have property with key `org`)

- **returned errors can be handled (e.g., set value to NULL)**
- **or can be avoided using filters**

SQL/JSON Path Language Member Access (X)

Example (*strict* mode): Access a property that does not exist for all array entries (with filters)



SQL/JSON Path Language Member Access (XI)

Example (*lax* mode): Use **wildcard** to access properties

JSON

```
{    "authors": [ {   "name": "S. Ruvimov",
                      "org":  "Div. of Mater. Sci (...)" },
                  { "name": "Z. Liliental-Weber" }
                ]
}
```

SQL/JSON Path Language

```
lax $.authors.*
```

...

JSON

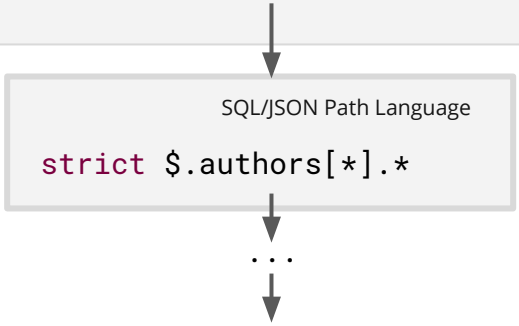
```
[ "S. Ruvimov", "Div. of Mater. Sci (...)",
  "Z. Liliental-Weber" ]
```

SQL/JSON Path Language Member Access (XII)

Example (*strict* mode): Use **wildcard** to access properties

JSON

```
{    "authors": [ {   "name": "S. Ruvimov",
                      "org":  "Div. of Mater. Sci (...)" },
                  { "name": "Z. Liliental-Weber" }
                ]
}
```



JSON

```
[ "S. Ruvimov", "Div. of Mater. Sci (...)",
  "Z. Liliental-Weber" ]
```

Array Element Access

SQL/JSON Path Language Array Element Access

Element access via `[ ]` (squared brackets) evaluation

Element access via comma-separated list of subscripts by mixing:

- single element index, e.g., `[0, 1, 2]`
- index range via to keyword, e.g., `[23 to 42]`
- special keyword `last` to refer to last element in array

Notes on array access

- For SQL/JSON Path Language, arrays start at index `0` (0-relative) in contrast to SQL
- Non-numeric subscripts result in error condition, e.g., `["42"]`

Mode differences for indexes outside bounds

- **In *strict* mode** returns an error condition
- **In *lax* mode** illegal indexes are ignored

SQL/JSON Path Language Array Element Access

Evaluation semantics of element access via `[ ]`

1. **Operator evaluation**
   Results in sequence of SQL/JSON items
2. **(a) In *strict* mode**
   Each SQL/JSON item in sequence must be of type SQL/JSON array. Otherwise, error.
   (b) In *lax* mode
   Each SQL/JSON item in sequence not of type SQL/JSON array is wrapped in array of size 1.
3. **Element fetch by index and concatenation**
   a. Index enumeration for each x in $[x_0, x_1, x_2, \ldots]$ for array A
      i. array index is expanded to final subscripts set L
         - if x is number n — L contains one element, n
         - if x is range n **`to`** m — L contains integers n, $n+1$, …, $m-1$, m
         - if x is **`last`** — L contains one element, (array size of A) - 1
      ii. results in SQL/JSON sequence S_x of elements in A having index in L (preserving order)
   b. All SQL/JSON sequences S_x with x in $[x_0, x_1, x_2, \ldots]$ are concatenated (preserving order)

SQL/JSON Path Language Array Element Access

Example (*lax* mode): Array element access (based on example from [ISO-SQL] p. 75)

JSON

```
{    "sensors":
     {
          "A": [10, 11, 12, 13, 15, 16, 17],
          "B": [20, 22, 24],
          "C": [30, 33]
     }
}
```

SQL/JSON Path Language

```
lax $.sensors.*[0, last, 2]
```

...

JSON

```
[ [10,17,12], [20, 24, 24], [30, 33]]
```

SQL/JSON Path Language Array Element Access

Example (*lax* mode): Array element access with **wildcard** (based on example from [ISO-SQL] p. 76)

JSON

```
{
    "x": [12, 30],
    "y": [8],
    "z": ["a", "b", "c"]
}
```

SQL/JSON Path Language

```
lax $.*[1 to last]
```

Evaluation of `lax $.*`

```
[12,30], [8], ["a", "b", "c"]
```

Evaluation of `[1 to last]`

```
30, (none), "b", "c"
```

JSON

```
[ 30, "b", "c"]
```

Item Functions

SQL/JSON Path Language Item Functions (I)

Higher-order built-in functions mapping SQL/JSON items to SQL/JSON items.
Typically invoked over a SQL/JSON sequence.

`type()`

Returns a string representation of the type of the SQL/JSON item *x* on which `type()` is invoked.

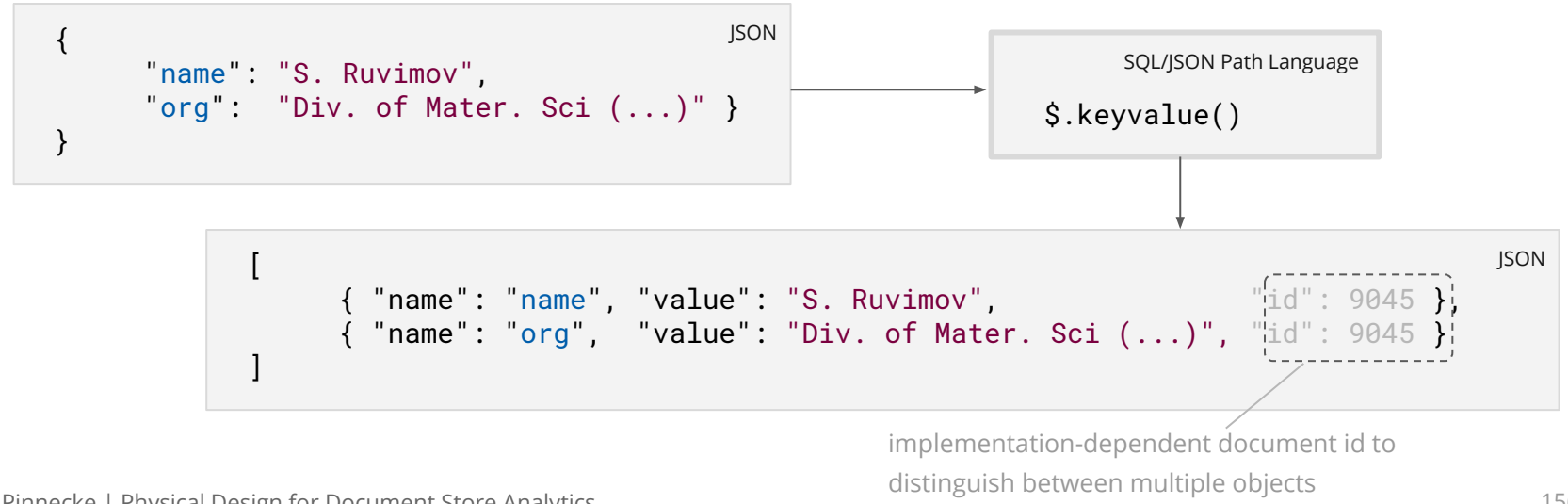
Input, *x* is SQL/JSON	**Output**
• null	`"null"`
• True, False	`"boolean"`
• numeric	`"number"`
• character string	`"string"`
• array	`"array"`
• object	`"object"`
• datetime	`"date", "time without time zone",...`

SQL/JSON Path Language Item Functions (II)

Higher-order built-in functions mapping SQL/JSON items to SQL/JSON items.
Typically invoked over a SQL/JSON sequence.

`keyvalue()`

Returns any SQL/JSON object (of unknown schema) to SQL/JSON sequence of objects with known schema. Useful for data exploration.



SQL/JSON Path Language Item Functions (III)

Higher-order built-in functions mapping SQL/JSON items to SQL/JSON items.
Typically invoked over a SQL/JSON sequence.

Additional functions

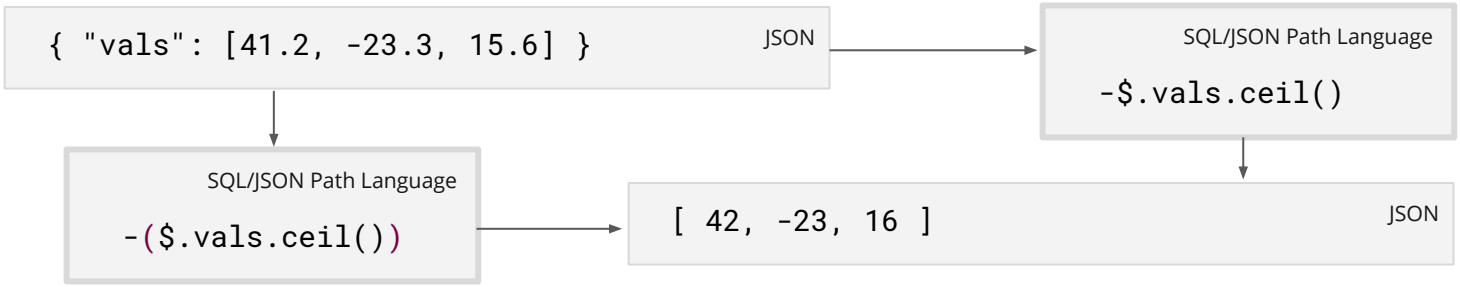
`size()`	returns number of elements in array, or 1 if object or scalar
`double()`	converts string or numeric value to numeric value
`ceiling()`	least integer greater than or equal to input numeric value
`floor()`	greatest integer less than or equal to input numeric value
`abs()`	non-negative of input numeric value ignoring the sign
`datetime()`	converts string to datetime typed value (mainly for comparison in predicates)

Arithmetic Expressions

SQL/JSON Path Language Arithmetic Expr. (I)

Built-in arithmetic operators

- Unary — Prefix operations *iterating* over a (numeric) SQL/JSON *sequence*
  - `+` (value)
  - `-` (negate)



Note Precedence of accessor binds more tightly than unary operators

SQL/JSON Path Language Arithmetic Expr. (II)

Built-in arithmetic operators

- Binary Infix operators between *two scalar values*
  - `+` (addition)
  - `-` (subtraction)
  - `*` (multiplication)
  - `/` (division)
  - `%` (modulus)

Filter Expressions

SQL/JSON Path Language Filter Expr. (I)

Filter expression are used to remove elements not satisfying predicate.

Example — SQL/JSON Path Language

```
lax $ ? (@.pay/@.hours > 9)
```

- **The `?` symbol**
  - Filter is expressed with a (parenthesized) predicate, starting with `?`
  - Various built-in predicates, such as *greater* comparison `>` (see next slide)
- **The `@` variable**
  - A special variable used to refer to current element in a sequence
  - When predicates are nested, `@` refers to innermost one

SQL/JSON Path Language Filter Expr. (II)

Notes on behavior and characteristics of filter expressions

Ternary logic	predicates evaluate either to `true`, `false`, or `unknown` (null)
Not assignable	predicates are not expressions in SQL/JSON path language
Items are not predicates	to verify `"b": true`, use `@.b == true` rather than `@.b`
SQL/JSON null compare	`null == null` evaluates to `true` (rather to *unknown* as in SQL)
Error handling	predicates evaluate to `unknown` if error (e.g., type mismatch), and the resulting SQL/JSON sequence is empty

SQL/JSON Path Language Filter Expr. (III)

Evaluation semantics

1. **Unwrapping of operand** (lax mode only)
   Any array **[** $x_0, x_1, \ldots, x_n$ **]** in the operand is unnested to $x_0, x_1, \ldots, x_n$
2. **Predicate evaluation**
   Predicate is evaluated for each SQL/JSON item in the sequence
3. **Resultset construction**
   SQL/JSON items for which the predicate evaluates to **`true`** are returned

SQL/JSON Path Language Filter Expr. (IV)

Ternary Truth Logic Tables

- Boolean operators (`&&`, `||`, and `!`) result in a truth value
  - **`true`**, **`false`**, and **`unknown`**

	Result of `&&`			Result of `||`			Result of `!`
value	**true**	**false**	**unknown**	**true**	**false**	**unknown**	**NOT** *value*
true	true	false	unknown	true	true	true	false
false	false	false	false	true	false	unknown	true
unknown	unknown	false	unknown	true	unknown	unknown	unknown

SQL/JSON Path Language Filter Expr. (V)

Built-in predicates

○ Comparisons	relational predicates
○ String matching	regular expression matching (**`like_regex`**)
○ Existence check	predicate to check whether a key exists (**`exists`**)
○ Prefix string match	test if string starts with another (**`starts with`**)
○ *null* (“unknown”) check	test if path results in *unknown* value (**`is unknown`**)

Comparison Predicates (I)

Example — SQL/JSON Path Language

```
lax $ ? (@.n_citations == 42)
```

- **Semantics.** Compares sequences (e.g., `n_cirations`) to constants (e.g., `42`) or sequences

`==`	equality	`<=`	less than or equal to
`!=` `<>`	inequality	`>`	greater than
`<`	less than	`>=`	greater than or equal to

- **Existential semantics**: Comparison of two sequences S_1 and S_2 computes the cross (cartesian) product $S_1 \times S_2$ (each item of S_1 is compared to each item in S_2)
  - **Evaluation.** Predicate φ (equality, less than, ...) results in
    - **`unknown`** (**`null`**) if one pair *(x, y)* of in $S_1 \times S_2$ is *not* comparable
      - e.g., *x* is boolean and *y* is number
      - lax mode: maybe **`true`** in some cases
    - **`true`** if any pair is comparable and satisfy the criteria
      - *x, y* of same type + for all φ(x,y)
    - **`false`** else

Comparison Predicates (II)

- Semantic differences compared to…
  - … JavaScript
    - `==` and `!=` (`<>`) predicates have same precedence
    - no casting across types (e.g, **`true`** `==` `1` results *not* in **`true`**)
    - no comparison of arrays and object to anything else (cf. unnesting in *lax* mode)
  - … SQL
    - SQL/JSON **`null`** `==` **`null`** results in **`true`** (rather than **`null`** as in SQL) !

String Matching Predicate

Example — SQL/JSON Path Language

```
lax $ ? (@.title like_regex regex)
```

- **Semantic.** Performs a pattern matching to a sequences (e.g., values for `title`) given a (SQL) regular expression *regex*
- **Evaluation.** Like *comparison* predicates, existential semantics is used

Prefix String Matching Predicate

Example — SQL/JSON Path Language

```
lax $ ? (@.authors.name starts with prefix-string)
```

- **Semantic.** Tests if first operand (e.g., sequences with values for `authors.name`) starts with a given string *prefix-regex*
- **Evaluation.** Like *comparison* predicates, existential semantics is used

Notes. `starts with` is equivalent to range comparison of strings

```
@.authors.name starts with "Pinn"  ≍
@.authors.name >= "Pinn" && @.authors.name < "Pino"
```

Existence Check Predicate

Example — SQL/JSON Path Language

```
lax $ ? (exists (@.title))
```

- **Semantic.** Tests if path has one or more items (i.e., if key exists for object at hand)
- **Evaluation.** After evaluation of the path (e.g., `.title`) for the current element in the sequence, the `exists` predicate results in
  - `unknown` (`null`) — if there is any error (e.g., no such key)
  - `false` — if the path is an empty sequence
  - `true` — else

Notes. `exists` predicate can be used to limit to elements having a specific key to avoid path errors in *strict* mode (see **member access via `.` (dot) evaluation semantics** from before)

Null Check Predicate

Example — SQL/JSON Path Language

```
lax $ ? (exists (@.title) is unknown)
```

- **Semantic.** Tests if a boolean condition results in `unknown` (e.g., `.title` does not exists)

  Notes. `is unknown` predicate can be used to find anomalous items, such as objects with missing keys or with wrong typing.

Summary

JSON Documents in Relational Systems

Summary JSON Documents in Rel. Systems

JSON Support in Relational Database Systems

- Overview on relational database systems supporting JSON
- JSON support in SQL Server 2016+ - import, handling, and JSON Path Expressions
- JSON in SQL:2016 Standard
  - Validation functionality (**`is [not] json`**)
  - Construction functionality (**`json_object`**, **`json_objectagg`**, **`json_array`**, **`json_arrayagg`**)
  - Query functions (**`json_exists`**, **`json_value`**, **`json_query`**, **`json_table`**)

SQL/JSON Path Language

- Architecture and embedding into SQL
- Path modes (*strict* and *lax*) - purpose and differences
- Data model, terms, mappings, SQL/JSON item, SQL/JSOM sequence
- Language Syntax and semantics
  - Variables (**`$`** and **`$<name>`**), member access (**`.`**) and array element access (**`[ ]`**)
  - Item functions (e.g., **`type()`**, or **`keyvalue()`**) and arithmetic expressions
  - Filter expressions (? and @, built-in predicates, evaluation semantics)

OTTO VON GUERICKE UNIVERSITÄT MAGDEBURG

Summary

What you've Learned (I)

Semi-structured data, arguments and implications

- Schema is not known in advance, or evolves heavily
- Database normalization is not required, or optional
- Application scenarios and use cases

Overview of database systems, and rankings

- Top-5 data models & trends
- Top-5 document stores

Document Database Model

- Fundamental terms (document, collection)
- Document collection vs tuples in tables
- JavaScript Object Notation (JSON): scoping, history, syntax
- JSON Schema to verify a document against a schema
- JSON Pointer to refer to specific value within a document

What you've Learned (II)

Document Stores Overview and Comparison

- Storage engine comparison - Append-Only vs Update-In-Place
- Different record formats and record organizations - JSON database vs BSON collections
- Query formulation, query language and database communication

CRUD (Create, Read, Update, Delete) Operations in mongoDB and CouchDB

- creation of databases, insertion of documents
- querying documents with filter operators, dot-notation, projection, sorting,...
- document identity (and for CouchDB revision management)
- aggregation query expression (and for CouchDB design documents)
- modification and deletion of databases and documents
- MapReduce as model and framework, usage and extensions in mongoDB vs CouchDB

What you've Learned (III)

Insights into one Append-Only and one Update-In-Place storage engine

- Database modifications and what happens underneath
- Document identity (document id), revision control and its application in CouchDB
- Multi-version management in CouchDB and MongoDB
- Discussion of pros and cons
- Insights into key properties of WiredTiger (MongoDBs storage engine)

Physical Record Organization

- Overview on representation formats for JSON-like records
- Key properties and example for Plain-Text JSON, UBJSON, BSON & CARBON
- CARBON archive file overview, complexity comparisons

What you've Learned (IV)

JSON Support in Relational Database Systems

- Overview on relational database systems supporting JSON
- JSON in SQL:2016 Standard
  - Validation functionality (**`is [not] json`**)
  - Construction functionality (**`json_object`**, **`json_objectagg`**, **`json_array`**, **`json_arrayagg`**)
  - Query functions (**`json_exists`**, **`json_value`**, **`json_query`**, **`json_table`**)

SQL/JSON Path Language

- Architecture and embedding into SQL
- Path modes (*strict* and *lax*) - purpose and differences
- Data model, terms, mappings, SQL/JSON item, SQL/JSOM sequence
- Language Syntax and semantics
  - Variables (**`$`** and **`$<name>`**), member access (**`.`**) and array element access (**`[ ]`**)
  - Item functions (e.g., **`type()`**, or **`keyvalue()`**) and arithmetic expressions
  - Filter expressions (? and @, built-in predicates, evaluation semantics)

OTTO VON GUERICKE UNIVERSITÄT MAGDEBURG

Final Words

Contribute to NG5/CARBON

Running Projects

Wire-Speed String Encoding for Main-Memory Databases (Individual Project)
SIMD Acceleration and Optimized Search in Libcarbon's multi-threaded string dictionary

Key-Based Self-Driven Compression in Columnar Binary JSON (Master's Thesis)
Key-domain-sensitive application of compression techniques in CARBONs string table with decision component to choose best fitting compression combination.

Open Projects (I)

AutoScale: Self-Driven Bucket-Scaling in Parallel String Dictionaries (Individual Project)
Design and implementation of a decision component to determine best number of buckets used in our parallel string dictionary.

AutoThreads: Smart Thread Spawning in Parallel String Dictionaries (Individual Project)
Design and implementation of a decision component to determine best number of threads to be used in our parallel string dictionary.

Json2Carbon: Improve Conversion Time from JSON to CARBON (Thesis)
Profile current implementation to find bottleneck in multi-step conversion routine, design and implementation new concepts, improve existing ones.

Carbon2Json: Improve Conversion Time from CARBON to JSON (Team Project)
Profile current implementation to find bottleneck in conversion routine, design and implementation an improved conversion routine.

Open Projects (II)

ReadOpt: Improve “Read-Optimization” Mode Execution for CARBON Archives (Thesis)
During conversion from JSON to CARBON, a special “read-optimized” option can be set that roughly performs an additional sorting. The current implementation is a proof-of-concept (by using clibs *qsort*). This thesis is about efficient sorting during conversion using modern hardware.

TransformOpt: Improve “Transformation Pipeline” for CARBON Conversions (Thesis)
During conversion from JSON to CARBON, a multi-stage transformation pipeline is entered to transform a “key-value-pair” JSON to a columnar representation inside CARBON. The current implementation is a proof-of-concept (not cache efficient, simple lookups). This thesis is about improving the transformation pipeline by smartly re-engineering parts of the transformation pipeline, and by applying advanced algorithm.

Quality: Testing of Several Components in Libcarbon and NG5 (Software Project)
Design and implement unit and integration tests for several components in the library.

Open Projects (III)

Split&Merge: Efficient Splitting and Merging of CARBON Archives (Thesis)
Currently, CARBON archives are constructed from a user-empowered JSON collection and read-only afterwards. In preparation of physical optimizations (such as undo archiving) and defragmentation, archives must be splittable and mergabele. This thesis is about this actions.

StringIdRewrite: Embedding of String ID Resolution w/o Indexes in CARBON (Thesis)
In the current form, resolving a fixed-length string reference in a CARBON archives - in case of a cache miss - requires to resolve the reference (string id) to the offset inside the string table on disk. This thesis is about rewriting archives by replacing string ids by their offset.

FastParse: Parallel JSON Parsing in Main Memory Databases (Individual Project)
To convert JSON files to CARBON files, the currently JSON parser works quite good. However, the parser is strictly sequential executed. Without multi-threading, parsing does not run at fullspeed as required for 1^+ GB JSON files. This project is about a concept, implementation and evaluation of parallel JSON parsing.

Open Projects (IV)

GeoJSON: Add Support of GeoJSON to CARBON Archives (Thesis)
Currently, CARBON archives do not support JSON arrays of JSON arrays. As a consequence, vector data or spatial data (such as GeoJSON) cannot be converted into CARBON archives. This thesis is about removing the restriction “no arrays of arrays” for CARBON archives.

JSON Check Tool as Separate Tool (Software Project)
Currently, in the CARBON Tool (carbon-tool) there is a sub module to check whether a particular JSON file is parsable and satisfies the criteria for conversion into CARBON archives (checkjs). Since this logic is shared with the BISON Tool (bison-tool), the task is to move the module in carbon-tool to a dedicated new tool called checkjs.

You didn’t find the right project but you have an idea or special interest?
Let me know!