Multi-Model Data Management and System Integration

NDBI040 - Modern Database Systems

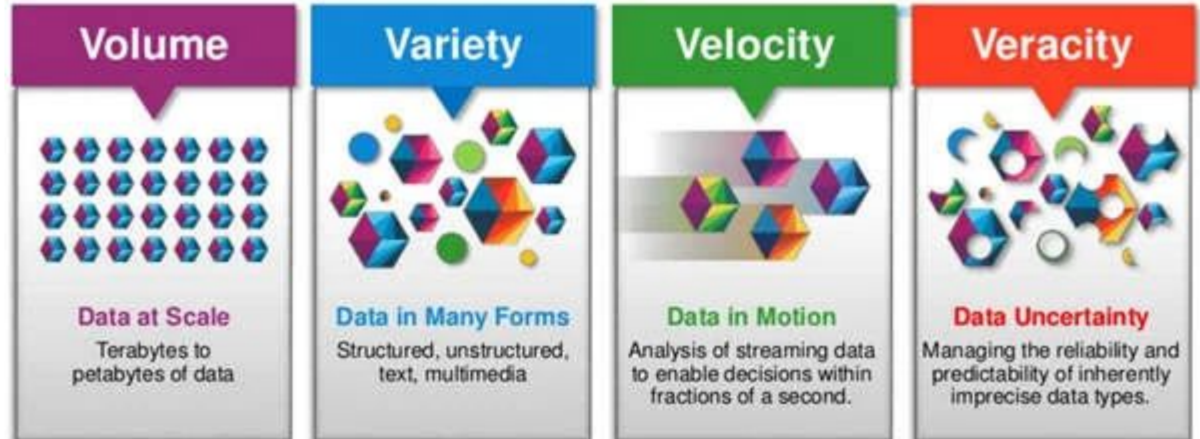
I. Holubová, P. Koupil

irena.holubova@matfyz.cuni.cz
pavel.koupil@matfyz.cuni.cz

https://www.ksi.mff.cuni.cz/~holubova/NDBI040/

Big Data V-Characteristics

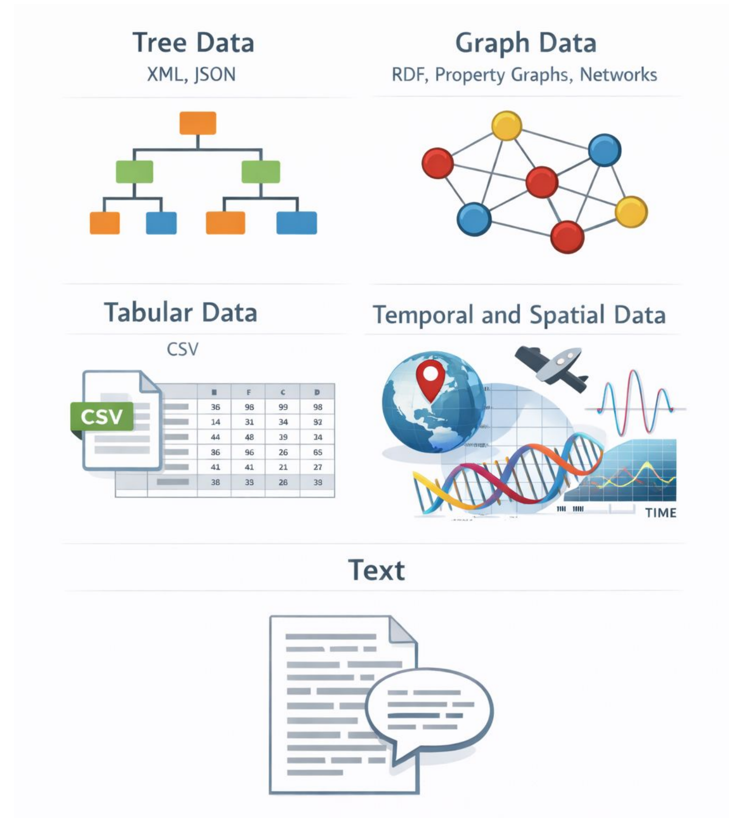
- Volume – scale
- Variety – complexity
- Velocity – speed
- Veracity – uncertainty
- Value
- Validity
- Volatility
- …



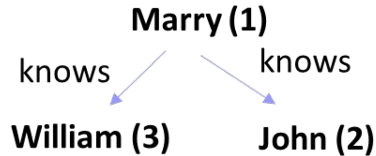
https://www.researchgate.net/figure/The-4-Vs-big-data-properties-volume-variety-velocity-veracity-9_fig1_338516812

Challenge: Data Variety

- Tree data
  - XML, JSON
- Graph data
  - RDF, property graphs, networks
- Tabular data
  - CSV
- Temporal and spatial data
- Text
- …



Example of Multi-Model Data



Social network graph

```
{ "Order_no":"0c6df508",
  "Orderlines": [
   { "Product_no":"2724f"
     "Product_Name":"Toy",
     "Price":66 },
   { "Product_no":"3424g",
     "Product_Name":"Book",
     "Price":40 } ]
}
```

Order JSON document

```
"1" -- > "34e5e759"
"2"-- > "0c6df508"
```

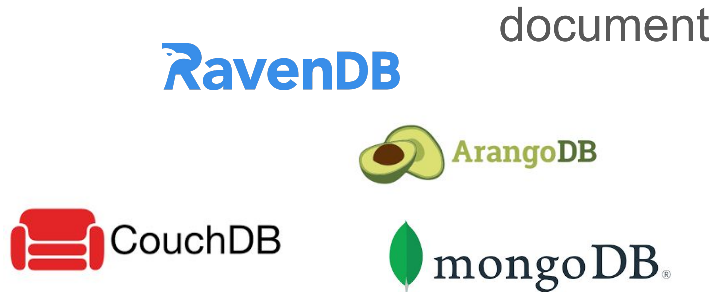
Key/value pairs (Customer_ID , Order_no)

Customer relation

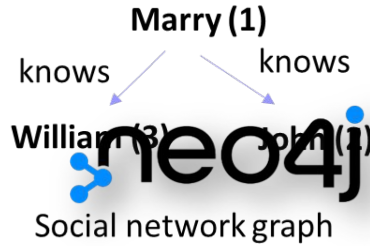
Customer_ID	Name	Credit_limits
1	Mary	5,000
2	John	3,000
3	William	2,000

Originally: different systems for different models

NoSQL DBMS



Example of Multi-Model Data



Social network graph

```
{ "Order_no":"0c6df508",
  "Orderlines": [
   { "Product_no":"2724f"
     "Product_Name":"Toy",
     "Price":66 },
   { "Product_no":"3424g",
     "Product_Name":"Book",
     "Price":40 } ]
}
```

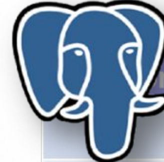
Order JSON document

"1" -- > "34e5e759"

"2"-- > "0c6df508"

Key/value pairs
(Customer_ID , Order_no)

Customer relation



…mer_ID	Name	Credit_limits
1	Mary	5,000
2	John	3,000
3	William	2,000

Polyglot Persistence

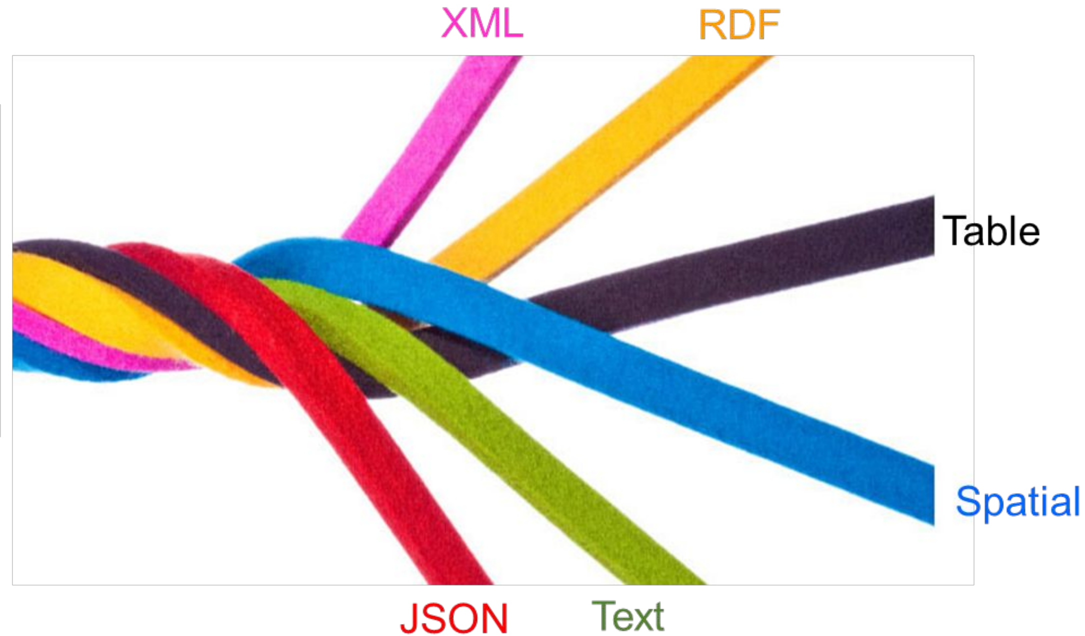
- Idea: Use the right tool for the job
- Document store → for structured data with differences
- Graph store → for relations
- Key/value store → for simple managed structure

But integration becomes hard…

Pros and Cons of Polyglot Persistence

- Pros:
  - handles multi-model data
  - helps apps scale
  - rich experience, mature systems
    - standards
- Cons:
  - requires integration expertise
  - developers must learn multiple DBs
  - challenge of cross-model queries and transactions

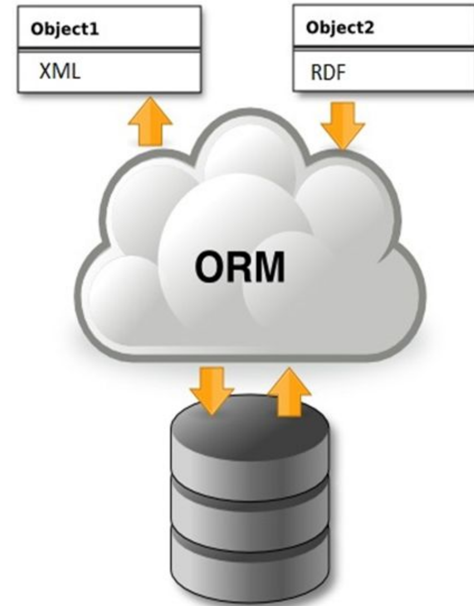
Multi-Model Databases



Designed to support multiple data models against a single, integrated backend

Multi-Model Databases Are Not New

- Its roots can be traced back to object-relational database systems (ORDBMS)
  - Extended the relational model with user-defined types, functions, indexes, …
- Moved beyond a single fixed data model



More DBMSs Become Multi-Model

Convergence trend: leading DBMSs gradually add support for multiple models

- relational systems ⟶ add JSON/XML
- document systems ⟶ add joins and graph-like operations
- graph systems ⟶ support nested attributes and documents

Rank			DBMS	Database Model	Score		
Apr 2026	Mar 2026	Apr 2025			Apr 2026	Mar 2026	Apr 2025
1.	1.	1.	Oracle	Relational, Multi-model	1157.93	-24.53	-73.12
2.	2.	2.	MySQL	Relational, Multi-model	857.69	-0.65	-129.41
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	702.08	-9.40	-82.93
4.	4.	4.	PostgreSQL	Relational, Multi-model			10
5.	5.	5.	MongoDB	Document, Multi-model			03
6.	6.	6.	Snowflake	Relational			28
7.	7.	↑12.	Databricks	Multi-model			19
8.	8.	↓7.	Redis	Key-value, Multi-model			58
9.	9.	9.	IBM Db2	Relational, Multi-model	113.25	+1.87	-12.79
10.	↑11.	↑11.	Apache Cassandra	Wide column, Multi-model	102.62	+0.74	-5.59
11.	↓10.	↓8.	Elasticsearch	Multi-model	99.51	-4.07	-28.57
12.	12.	↓10.	SQLite	Relational	95.90	-0.08	-18.19
13.	13.	↑14.	MariaDB	Relational, Multi-model	86.41	-0.59	-7.93

Relational DBMS
Document store
Graph DBMS
Spatial DBMS
Vector DBMS

https://db-engines.com/en/ranking

Pros and Cons of Multi-model Databases

- Pros:
  - handle multi-model data
  - one fault-tolerant system
  - data consistency
  - unified query language
- Cons:
  - complex
  - immature
  - no standards

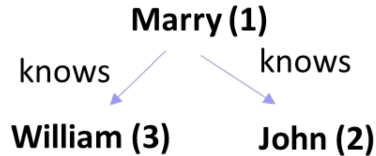
Example

- open-source
- **document-first** multi-model
  - documents ⟶ graphs, key/value
- stores all data as documents
- vertices and edges are documents
- unified query language (AQL)

- open-source
- **graph-first** multi-model
  - graphs (objects) ⟶ documents
- records = vertices / edges / documents
- supports embedding and references
- SQL-like query language + graph traversals

Example of Multi-Model Data

Return all product numbers ordered by a friend of a customer whose credit limit is greater than 3000.



Social network graph

```
{ "Order_no":"0c6df508",
  "Orderlines": [
   { "Product_no":"2724f"
     "Product_Name":"Toy",
     "Price":66 },
   { "Product_no":"3424g",
     "Product_Name":"Book",
     "Price":40 } ]
}
```

Order JSON document

"1" -- > "34e5e759"

"2"-- > "0c6df508"

Key/value pairs
(Customer_ID , Order_no)

Customer relation

Customer_ID	Name	Credit_limits
1	Mary	5,000
2	John	3,000
3	William	2,000

Return all product numbers ordered by a friend of a customer whose credit limit is greater than 3000.

```
LET customerIDs = (
  FOR customer IN Customers
    FILTER customer.CreditLimit > 3000
    RETURN customer.id
)

LET friendIDs = (
  FOR customerID IN customerIDs
    FOR friend IN 1..1 OUTBOUND customerID Knows
      RETURN friend.id
)

FOR friendID IN friendIDs
  FOR order IN 1..1 OUTBOUND friendID Customer2Order
    RETURN order.Orderlines[*].Product_no
```

Return all product numbers ordered by a friend of a customer whose credit limit is greater than 3000.

```
SELECT expand( out("Knows").out("orders").orderlines.product_no )
FROM Customers
WHERE CreditLimit > 3000
```

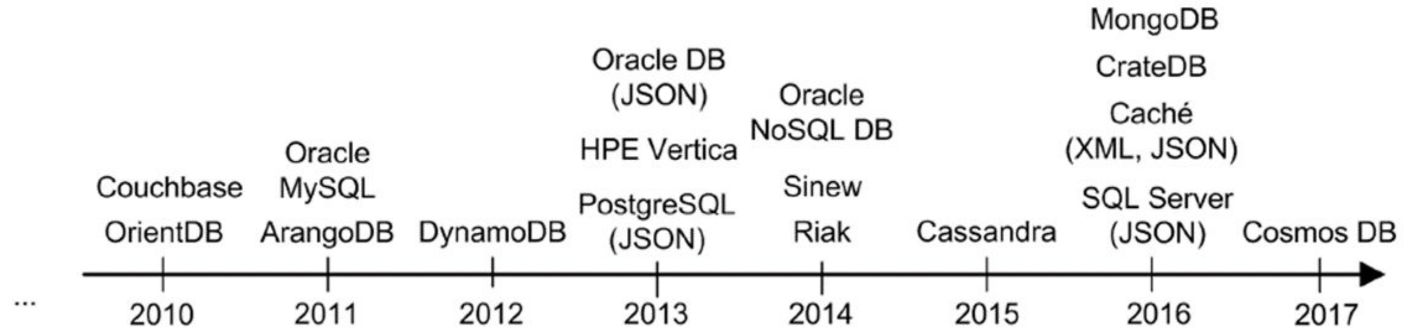
Classification of Multi-model Systems

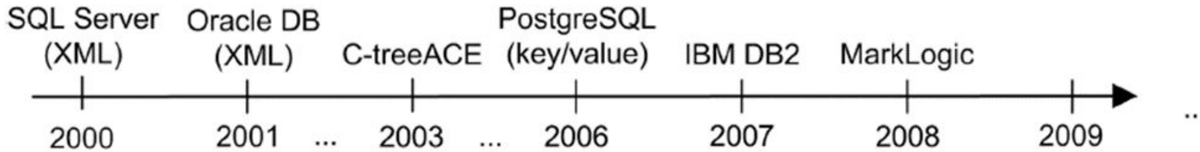
According to the original model:

- Relational → PostgreSQL, SQL Server, IBM DB2, Oracle DB, MySQL
- Document → ArangoDB, Couchbase, MarkLogic, MongoDB, Cosmos DB
- Graph → OrientDB
- Key/value → Riak, Oracle NoSQL DB
- Column → Cassandra, CrateDB, DynamoDB, HPE Vertica

Different roots → different multi-model behavior

Early Evolution of Multi-Model Systems





How Systems Become Multi-Model

- New storage strategy for a new model
  - e.g., Relational systems adding native support for nested data
    - SQL/XML, SQL/JSOn standards
- Extend existing storage strategy
  - e.g., ArangoDB - nodes and edges stored as document collections
- New query interface
  - e.g., MarkLogic - stores JSON data in the same way as XML data
- No storage change
  - Formats simpler than the original one

Types of Cross-Model Transitions

1. Inter-model references

- links via IDs across models
- graph traversal over external data

✔ flexible, normalized
✖ requires joins / traversals

2. Model embedding

- relational data stored as nested JSON
- denormalization into document model

✔ fast reads, locality
✖ duplication, harder updates

3. Cross-model redundancy

- same data stored in multiple models

✔ performance, fewer joins
✖ consistency, update anomalies

Reference = integration via links
Embedding = integration via structure
Redundancy = integration via duplication

Practical Integration Challenges I

- Cross-model query processing
  - joins across models (graph ↔ document ↔ relational)
  - different execution strategies
  - no unified optimizer
- Data consistency
  - redundancy → update anomalies
  - embedding → duplicated state
  - references → distributed updates
- Schema & model evolution
  - JSON → flexible schema
  - relational → strict schema
  - graph → evolving structure

Practical Integration Challenges II

- Indexing & optimization
  - B-tree (relational)
  - inverted / JSON index (document)
  - traversal optimization (graph)
- Transactions across models
  - ACID vs eventual consistency
  - cross-model transactions expensive
- Developer complexity
  - multiple query languages
  - different APIs
  - mental model mismatch

Integration is not only about data — it is about combining incompatible assumptions

```
CREATE TABLE customer (
  id      INTEGER PRIMARY KEY,
  name    VARCHAR(50),
  address VARCHAR(50),
  orders  JSONB
);
```

```
INSERT INTO customer
VALUES (1, 'Mary', 'Prague',
 '{"Order_no":"0c6df508",
   "Orderlines":[
     {"Product_no":"2724f", "Produ
     {"Product_no":"3424g", "Produ
 }');
INSERT INTO customer
VALUES (2, 'John', 'Helsinki',
 '{"Order_no":"0c6df511",
   "Orderlines":[
     { "Product_no":"2454f", "Product_Name":"Computer", "Price":34 }]
 }');
```

```
SELECT name,
  orders->>'Order_no' as Order_no,
  orders#>'{Orderlines,1}'->>'Product_Name as Prod_name
FROM customer
where orders->>'Order_no' <> '0c6df511';
```

id integer	name character varying (50)	address character varying (50)	orders jsonb
1	Mary	Prague	{"Orderlines":[{"Price":66,"Product_Name":"Toy","Product_no":"2724f"},{"Price":40,"Product_Name":...
2	John	Helsinki	{"Orderlines":[{"Price":34,"Product_Name":"Computer","Product_no":"2454f"}],"Order_no":"0c6df511"}

```
SELECT orders[0]
FROM myspace.customer
WHERE id = 1;
```

```
create keyspace myspace
WITH REPLICATION = { 'class' : 'SimpleStrategy', 'replication_factor' : 3 };
CREATE TYPE myspace.orderline (
   product_no text,
   product_name text,
   price float
   );
CREATE TYPE myspace.myorder (
   order_no text,
   orderlines list<frozen <orderline>>
   );
CREATE TABLE myspace.customer (
   id INT PRIMARY KEY,
   name text,
   address text,
   orders list<frozen <myorder>>
   );
```

```
INSERT INTO myspace.customer JSON
' {"id":1,
   "name":"Mary",
   "address":"Prague",
   "orders" : [
   { "order_no":"0c6df508",
     "orderlines":[
     { "product_no" : "2724f",
       "product_name" : "Toy",
       "price" : 66 },
       { "product_no" : "3424g",
         "product_name" :"Book",
         "price" : 40 } ] } ]
  }';
```

XQuery:

```
let $product := fn:doc("/myXML1.xml")/product
let $order := fn:doc("/myJSON1.json")[Orderlines/Product_no = $product/@no]
return $order/Order_no

Result: 0c6df508
```

JavaSript:

```
declareUpdate();
 xdmp.documentInsert("/myJSON1.json",
 {
   "Order_no":"0c6df508",
    "Orderlines":[
     { "Product_no":"2724f",
       "Product_Name":"Toy",
       "Price":66 },
     {"Product_no":"3424g",
      "Product_Name":"Book",
      "Price":40}]
 }
 );
```

XQuery:

```
xdmp:document-insert("/myXML1.xml",
 <product no="3424g">
   <name>The King's Speech</name>
   <author>Mark Logue</author>
   <author>Peter Conradi</author>
 </product>
 );
```

```
CREATE CLASS orderline EXTENDS V
 CREATE PROPERTY orderline.product_no STRING
 CREATE PROPERTY orderline.product_name STRING
 CREATE PROPERTY orderline.price FLOAT

CREATE CLASS order EXTENDS V
 CREATE PROPERTY order.order_no STRING
 CREATE PROPERTY order.orderlines EMBEDDEDLIST orderline

CREATE CLASS customer EXTENDS V
 CREATE PROPERTY customer.id INTEGER
 CREATE PROPERTY customer.name STRING
 CREATE PROPERTY customer.address STRING

CREATE CLASS orders EXTENDS E

CREATE CLASS knows EXTENDS E
```

```
CREATE VERTEX order CONTENT {
    "order_no":"0c6df508",
     "orderlines":[
      { "@type":"d",
        "@class":"orderline",
        "product_no":"2724f",
        "product_name":"Toy",
        "price":66 },
      { "@type":"d",
        "@class":"orderline",
        "product_no":"3424g",
        "product_name":"Book",
        "price":40}]
 }
```

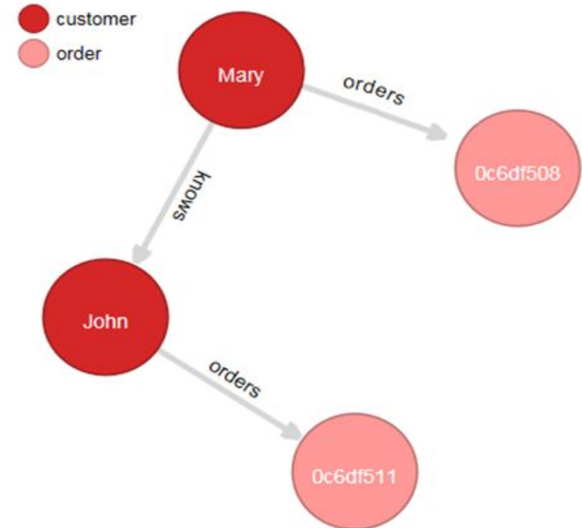
```
CREATE VERTEX order CONTENT {
    "order_no":"0c6df511",
     "orderlines":[
      { "@type":"d",
      "@class":"orderline",
      "product_no":"2454f",
      "product_name":"Computer",
      "price":34 }]
 }
CREATE VERTEX customer CONTENT {
    "id" : 1,
    "name" : "Mary",
    "address" : "Prague"
    }
CREATE VERTEX customer CONTENT {
    "id" : 2,
    "name" : "John",
    "address" : "Helsinki"
    }
```

```
SELECT out("knows").out("orders").orderlines.product_no
FROM customer
WHERE name="Mary";
```

```
CREATE EDGE orders FROM
    (SELECT FROM customer WHERE name = "Mary")
    TO
    (SELECT FROM order WHERE order_no = "0c6df508")

CREATE EDGE orders FROM
    (SELECT FROM customer WHERE name = "John")
    TO
    (SELECT FROM order WHERE order_no = "0c6df511")

 CREATE EDGE knows FROM
    (SELECT FROM customer WHERE name = "Mary")
    TO
    (SELECT FROM customer WHERE name = "John")
```



Multi-Model Database Challenges

Modeling & Representation

- Open data model
- Schema & model evolution
- References vs embedding vs redundancy

Storage & Indexing

- Multi-model index structures
- Different access patterns
  - Graph / JSON / table
- No universal index

Transactions & Consistency

- Multi-model transactions
- ACID vs eventual consistency
- Consistency across representations

Query & Processing

- Unified query language
- Cross-model query processing
- No universal optimizer

ArangoDB

- Open-source multi-model DBMS
  - Document-first system
- Supports:
  - document model
  - graph model
  - key/value access
- Stores data in collections
  - Graph data represented by vertex and edge collections
- One query language for multiple models: AQL

How ArangoDB Stores Data

- ArangoDB is document-first
  - Everything is stored in collections
  - A document is a JSON-like object
- Each document has system attributes:
  - `_key` = local identifier (within a collection)
  - `_id` = collection/`_key`
  - `_rev` = revision (changes on update)
- Graphs are built on top of documents:
  - vertex collections store vertices
  - edge collections store edges
- Edge documents contain:
  - `_from`
  - `_to`

AQL: Arango Query Language

- Declarative query language
  - Designed for JSON-like data
  - Works over documents and graphs
- Supports:
  - filtering (`FILTER`)
  - projection / return (`RETURN`)
  - aggregation (`COLLECT`, `AGGREGATE`)
  - sorting (`SORT`), limiting (`LIMIT`)
  - traversals
  - shortest path queries
  - ...

```
FOR x IN collection
FILTER ...
SORT ...
LIMIT ...
RETURN ...
```

AQL for Document Querying

collection: `people_doc`

```
{
  "_key": "u10",
  "id": 10,
  "given_name": "Alice",
  "family_name": "Brown",
  "email": "alice@example.com",
  "likes": [
    {"target": "databases", "weight": 5},
    {"target": "graphs", "weight": 3},
    {"target": "aql", "weight": 2}
  ]
}
```

```
{
  "_key": "u11",
  "id": 11,
  "given_name": "Bob",
  "family_name": "Miller",
  "email": "bob@example.com",
  "likes": [
    {"target": "graphs", "weight": 4},
    {"target": "ai", "weight": 2}
  ]
}
```

AQL for Document Querying

```
FOR p IN people_doc
  FILTER p.id == 10
  RETURN {
    person: {
      id: p.id,
      email: p.email,
      given_name: p.given_name,
      family_name: p.family_name
    },
    likes: p.likes
  }
```

```
LET me = FIRST(
  FOR p IN people_doc
    FILTER p.id == 10
    RETURN p
)
```

```
FOR l IN me.likes
  COLLECT target = l.target
  AGGREGATE score = SUM(l.weight)
  SORT score DESC
  LIMIT 10
  RETURN { target, score }
```

AQL for Graph Traversal

vertex collection: `people`

```
{
  "_key": "u10",
  "name": "Alice"
}
```

```
{
  "_key": "u11",
  "name": "Bob"
}
```

```
{
  "_key": "u12",
  "name": "Carol"
}
```

edge collection: `follows`

```
{
  "_key": "f1",
  "_from": "people/u10",
  "_to": "people/u11",
  "since": 2023,
  "strength": 0.9
}
```

```
{
  "_key": "f2",
  "_from": "people/u11",
  "_to": "people/u12",
  "since": 2024,
  "strength": 0.7
}
```

AQL for Graph Traversal

```
FOR v, e, p IN 1..2 OUTBOUND startVertex edgeCollection
RETURN ...
```

Traverse from `startVertex` along `edgeCollection`, following outgoing edges up to the given depth

- **`v` = current vertex**
- **`e` = the last edge**
- **`p` = the whole path**

```
FOR v, e, p IN 1..2 OUTBOUND DOCUMENT("people/u10") follows
  RETURN {
    vertex: v.name,
    via: e._key,
    path: p.vertices[*].name
  }
```

```
[
  { "vertex": "Bob",   "via": "f1", "path": ["Alice", "Bob"] },
  { "vertex": "Carol", "via": "f2", "path": ["Alice", "Bob", "Carol"] }
]
```

```
FOR v, e IN OUTBOUND SHORTEST_PATH DOCUMENT("people/u10") TO DOCUMENT("people/u12")
  follows
  RETURN v.name
```

```
["Alice", "Bob", "Carol"]
```

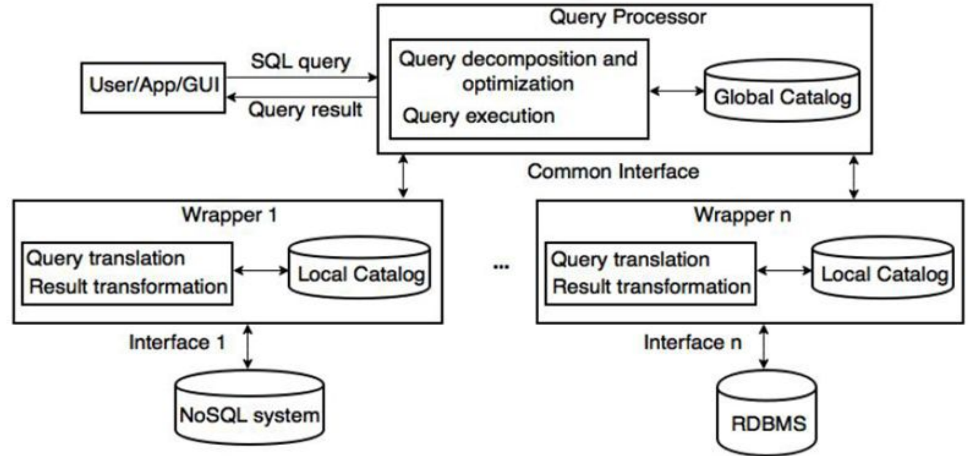
Two Integration Strategies for Multi-Model Data

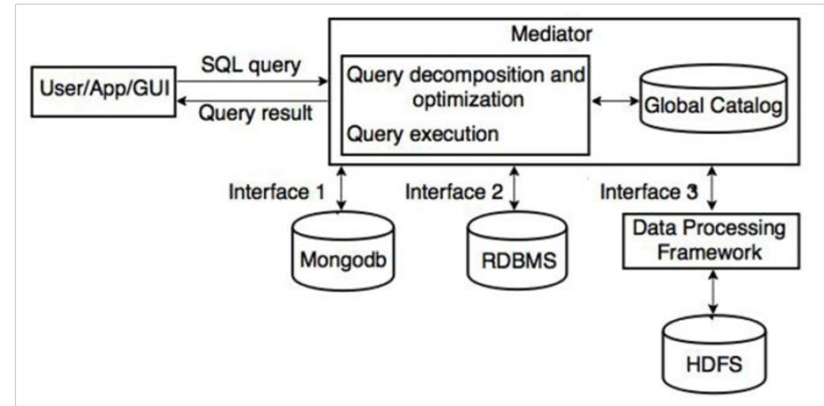
- **Multi-model databases**
  - One single, integrated backend
- **Polystores**
  - Multiple data storage technologies used jointly
  - Idea: Use the right tool for (each part of) the job
    - hierarchical data → document store
    - relationships → graph database
    - simple scalable acces → key-value store
    - full-text search → search engine
    - …

  and glue everything together

Types of Polystores

- **Loosely-coupled**
  - mediator-wrapper style
  - common interface
  - autonomy of local stores
- **Tightly-coupled**
  - direct use of local interfaces
  - better performance
  - lower autonomy
- **Hybrid**
  - combines both





Dimensions of Polystores

- Heterogeneity
  - different data models, different query languages, different engines
- Autonomy
  - local stores keep own interfaces and evolution
  - federation must respect local behavior
- Transparency
  - hide data location and movement
  - but still explain decisions when needed
- Flexibility
  - modular architecture
  - extensible interfaces and mappings
- Optimality
  - federated plans
  - data placement and migration matter

Mostly academic / research systems

System	Origin / Creators	Combines
BigDAWG	MIT + Intel + Brown University	PostgreSQL + SciDB + Accumulo
CloudMdsQL	INRIA / LIRMM (France)	relational + NoSQL stores
Myria	University of Washington	distributed analytics backends
Musketeer	University of Cambridge	multiple processing engines
QoX / Forward / SQL++	academic prototypes	heterogeneous stores

Related enterprise platforms

System	Origin	Focus
Denodo	Denodo Technologies	data virtualization
Teiid	Red Hat / JBoss	federation
Trino / Presto	Facebook / open source	SQL over many sources

Why Polystores Are Hard?

- Each engine has
  - own model
  - own query language
  - own optimizer
  - own consistency model
- Integration must bridge all layers
- Query processing:
  - parse query
  - split into subqueries
  - run in local engines
  - move partial data
  - join / merge results

Legacy of Polystores

- Polystores as a standalone category promised to:
  - connect multiple heterogeneous engines
  - send subqueries to the most suitable system
  - move data between systems when needed
  - hide heterogeneity behind a single integration layer
- This did not become mainstream, but ideas reappeared in cloud data platforms
  - unified access over multiple engines and sources
  - support for heterogeneous workloads under one platform
  - data movement via caching, replication, ETL / ELT
  - …

Polystore ideas survived more as platform architecture than as standalone DBMSs

References

- Neither Fish Nor Fowl: the Rise of Multi-model Databases. The 451 Group, 2013.
- J. Lu, Z. H. Liu, P. Xu, and C. Zhang. UDBMS: road to unification for multi-model data management. CoRR, abs/1612.08050, 2016
- J. Lu: Towards Benchmarking Multi-model Databases. CIDR 2017
- Kolev, B. et al.: Benchmarking Polystores: the CloudMdsQL Experience
- Kharlamov, E. et al.: A Semantic Approach to Polystores
- ArangoDB https://arango.ai/