

Querying, Extracting & Integrating Data

Faculty of Computer Science
Workflow Systems and Technologies

Interoperability

Motivational Example

Can we treat the web as a database?

If so, how can we query for specific data or information?

Data to Wisdom

Ackoff (1989):

- **Data:** symbols; result of observations.
- **Information:** Inferred from data by processing it into useful form; answers questions concerning *who, what, where, when*.
- **Knowledge:** Answers *how* something works. Acquired by transmission from another or by extracting it from experience; obtaining is learning.
- **Understanding:** Explains the *why* part.
- **Wisdom:** evaluated *understanding*, in terms of long- and short-range consequences.

Classification of Data

Unstructured Data

- Does not have clear, semantically overt, easy-for-a-computer structure (Manning et al., 2008)
- Typical examples include text, sound, images and videos.
- Internally data may be composed of structured building blocks, e.g.:
 - Text corpus consists of headings and paragraphs (explicit markup).
 - Images have distinct color channels and clear arrangement of pixels.
- *Search in unstructured data is more challenging, e.g.:*
 - *Search images which have tea cups and shortbread biscuits.*
 - *Retrieve documents concerning the Austrian general election of 1945.*

Classification of Data

Structured Data

- In contrast to unstructured data, structured data is strictly arranged with regards to a data schema (e.g. a relational schema).
- Lends itself to structured search against elements defined in the schema. Examples:
 - *Look up the details of a sales order with the order ID X.*
 - *Retrieve all purchase orders which include item Y.*
- SQL - Structured Query Language based on relational algebra

Classification of Data

Semi-Structured Data

- May not be constrained by a schema; if a schema is present it may only be loosely imposed.
- This form of flexibility can be highly desirable in certain scenarios:
 - E.g. NoSQL databases *do not enforce* predefined rigid schema.
 - Allows to grow documents and thus schema arbitrarily over time.
- Schema usually *inherent part of data*; thus it is "*self-describing*" (Buneman, 1997).
- *Structured retrieval (sometimes also referred to as semi-structured retrieval) of data or documents possible, e.g., via XPath*; however, requires knowledge about structure.

Data Integration

- Goal is to merge different data sources into a unified representation.
- Information systems (IS) are not always designed with integration in mind
- Why data integration is hard:
 - Incomplete or missing data
 - Varying data formats and data access interfaces
 - Schema heterogeneity
 - Different interpretation and representation of concepts
 - Data privacy and regulatory compliance
 - ...

Data Integration Levels

Ziegler and Dittrich (2007)

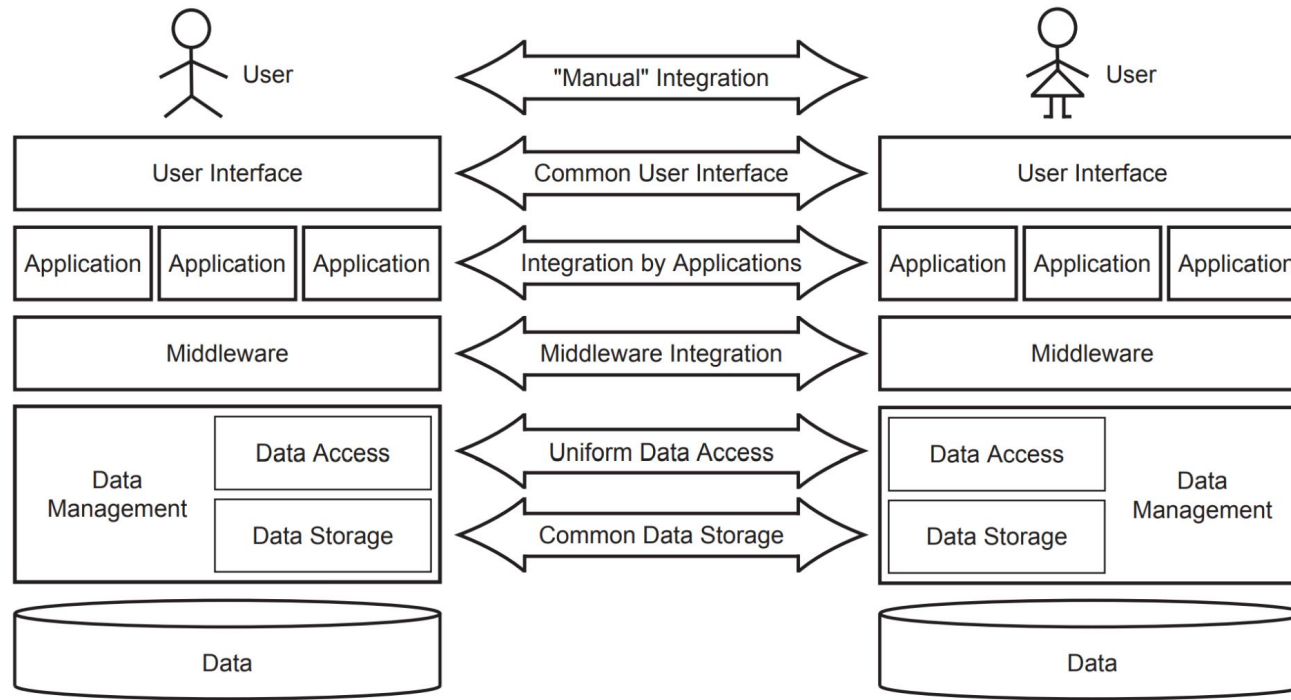


Fig. 1.1. General Integration Approaches on Different Architectural Levels

Data Integration Levels

- **Manual Integration** expects users to manually interact with relevant IS and integrate selected data. Requires ability to handle different user interfaces and query languages, including detailed knowledge on location, logical representation and semantics of data.
- **Common User Interface** provides a uniform user interface such as a web browser to relevant IS. However, data is still presented separately (just as with a search engine), thus requiring manual integration efforts by users.
- **Integration by Applications** utilises integration applications to access data from various sources and return the integrated result to the user or system. Can turn into complex components as systems and data formats increase.

Data Integration Levels

- **Integration by Middleware** focuses on using (a combination of) middleware solutions that can handle common integration problems. However, additional integration efforts are still necessary in applications.
- **Uniform Data Access** employs the data access layer for logical integration of data. Provides global applications with a unified global view of *physically distributed* data of local IS. Users are unaware of the physical location of the data or IS → *data virtualisation*. A global view with physical data integration is less feasible as integration occurs at runtime.
- **Common Data Storage** relies on physical integration of data. Data is directly transferred to a physical data storage. Applications must also be migrated to the new data storage when an old storage is retired.

Dimensions of Big Data Integration

- Scale of today's data...
 - *E.g., CERN's LHC produces 1 petabytes (PB) of data per second!**
 - *1PB = 1.000.000.000.000.000 bytes = 10^{15} bytes = 1000 terabytes*
- Goal is to **analyze** and **extract** value from data for data-driven decision making that impacts various aspects of society. (Dong et al., 2013)
- Value of data sources increases if they can be **related** and **unified** into *a shared representation*.

* (Galliard, 2017)

Dimensions of Big Data Integration

Key dimensions of big data integration (Dong et al., 2013):

- **Volume:** Data sources can contain huge volumes of data.
- **Velocity:** The growth in volume is determined by the *rate at which data is collected* and made available. Data sources are dynamic; they *evolve and increase* over time.
- **Variety:** Data sources can be heterogeneous at the *schema level* with regards to how data is structured but also in terms of how an entity is described at the *instance level*.
- **Veracity:** Quality of data sources may vary significantly (also when they are part of the same domain) in terms of *coverage, accuracy and timeliness* of the provided data.

Relational Databases

Relational database management systems (RDBMS) enforce a horizontal schema:

- Each column (attribute) of a relation (table) represents a specific data type.
- Each row represents a tuple of data elements.
- Relations can be related using attributes (typically primary keys).
- Structured Query Language (SQL) enables to query against relations using elements of schema.

SQL Schema

```
CREATE Table foo(  
    bar text,  
    primary key(bar)  
);
```

SQL Query

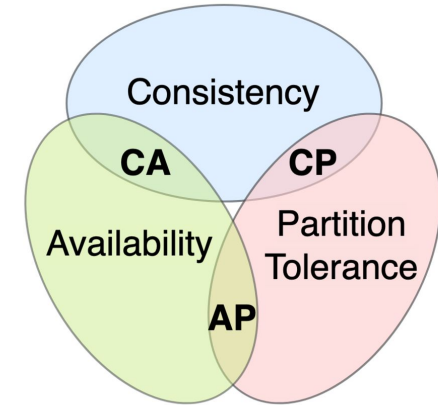
```
SELECT *  
FROM foo  
WHERE foo.bar = "foobar";  
ORDER BY foo.bar
```

Non-Relational Databases

- In addition to relational databases (DBs) a number of alternative database types are known under the umbrella term NoSQL (not only SQL). (Han et al., 2011)
- NoSQL databases can be classified into
 - Document databases (e.g. MongoDB)
 - Key-Value databases (e.g. Redis)
 - Column-oriented databases (e.g. Cassandra)
 - Graph databases (e.g. Neo4j)
 - (Object-oriented databases)
- Usually based on the **BASE** principle (basically-available, soft-state, eventual consistency) in contrast to the traditional **ACID** model (atomicity, consistency, isolation, durability) typical for relational DBs. Some also provide ACID properties.

CAP-Theorem

- Data is often scattered in physically remote storages.
- Quality of logical integration subject to the typical problems of distributed systems.
- CAP-Theorem states the properties consistency, availability and partition tolerance cannot be guaranteed at the same time in such circumstances.



XML

- Extensible Markup Language (XML)
- General purpose, simple and flexible text format
- Designed originally for large-scale electronic publishing (W3C, 2016)
- Supports many standards and tools
 - XML/EDIFACT for B2B transactions
 - WSDL (SOAP) for service descriptions
 - RDF/XML for semantic web
 - Processing languages (e.g. XSLT)

Example 1: EDIFACT source code [\[edit \]](#)

A name and address (NAD) segment, containing customer ID and customer address, expressed in EDIFACT syntax:

```
NAD+BY+CST9955::91++Candy Inc+Sirup street 15+Sugar Town++55555'
```

Example 2: XML/EDIFACT source code [\[edit \]](#)

The same information content in an XML/EDIFACT instance file:

```
<S_NAD>
  <D_3035>BY</D_3035>
  <C_C082><D_3039>CST9955</D_3039><D_3055>91</D_3055></C_C082>
  <C_C080><D_3036>Candy Inc</D_3036></C_C080>
  <C_C059><D_3042>Sirup street 15</D_3042></C_C059>
  <D_3164>Sugar Town</D_3164>
  <D_3251>55555</D_3251>
</S_NAD>
```


Data- and Document-centric XML Applications

Data-centric applications

- Typically highly structured data composed mostly of numeric and non-text (simple attribute-centric values) data.
- Querying against clearly defined structure using exact matches - structured retrieval.
- Examples: exporting data from relational database for data exchange.

```
<course name="interoperability">
  <topics count="2">
    <topic>Data-Centric Applications</topic>
    <topic>Document-Centric Applications</topic>
  </topics>
  <course_id>052512</course_id>
</course>
```

Data- and Document-centric XML Applications

Document-centric applications

- Suitable for publishing semi-structured text-centric documents.
- Typically for human consumption, however, also automatically processable within the structural constraints.
- Example: nesting catalogs in HTML (XML's origin lies in SGML)

```
<html>
<body>
  <h2>Interoperability</h2>
  <section>
    <p>This course is about <i>interoperability</i>.</p>
    <h4>Topics</h4>
    <ul>
      <li><a href="/document-centric">Document-Centric Applications</a></li>
      <li><a href="/data-centric">Data-Centric Applications</a></li>
    </ul>
  </section>
</body>
```

XML Basics

- XMLs are composed of elements and attributes.
- An element is represented by an opening and closing tag:
 - `<a>`
 - `<a/>` shorthand for an empty element.
- Elements can contain text and nest elements.
- Attributes are assigned as follows:
 - ``
 - ``
- A well-formed XML document may only have a single root element.
- `<!-- Comment syntax -->`

JSON

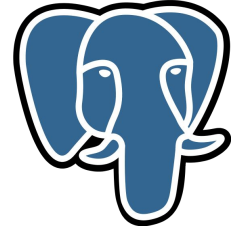
- (IETF 2017):
"JavaScript Object Notation (JSON) is a lightweight, text-based, language-independent data interchange format. It was derived from the ECMAScript Programming Language Standard. JSON defines a small set of formatting rules for the portable representation of structured data."
- Typically used for data-centric applications
- Most prominent example: data-centric web-services

JSON Basics

- JSON is represented by four primitive types (strings, numbers, booleans, and null) and two structured types (objects {} and arrays [])
- JSON documents can be deeply nested using objects.
- An object represents a hash of key and value pairs, arrays can hold arbitrary JSON types:
 - Object: {"key": 1, "2": "two"}
 - Array: ["a", 1.0, 1, true, false, null, {}, []]
- Keys must be of the type string, values can be any JSON type.
- A well-formed JSON may only contain a single primitive or structured type at the document's root level.

Structured Retrieval in Action

PostgreSQL



- (Object-)relational database
- Project POSTGRES started in 1986 at UC Berkeley
- In addition to the traditional primitive types normally associated with a relational database, document data (e.g. XML and JSON) and key-value pairs as a single attribute value are also supported.
- Supports data-centric and advanced text-centric search

SQL/XML and SQL/JSON

- SQL/XML and SQL/JSON are standardised by ISO/IEC 9075 and ISO/IEC 19075-6 respectively.
- Enable to process and extract relational data as XML and JSON
- Check the documentation of databases for the coverage of the features defined by the standards!
- For PostgreSQL see [SQL/XML](#) and [SQL/JSON](#).

Example Data

```
DROP TABLE IF EXISTS beings CASCADE;  
DROP TABLE IF EXISTS being_traits;
```

```
CREATE TABLE beings(  
  being_name      text,  
  power_level     bigint,  
  address         xml,  
  powerfoods      jsonb,  
  
  primary key (being_name)  
);
```

```
CREATE TABLE being_traits(  
  being_name      text,  
  trait_code      text,  
  trait_score     int,          -- 1 lowest and 5 highest score  
  
  foreign key (being_name) references beings(being_name),  
  primary key (trait_code, being_name)  
);
```

Example Data

insert into beings values

```
('Storm', 9900,  
  '<address><location city=""/><planet>Earth</planet></address>',  
  ' [{"food": "Oatmeal"}, {"food": "Grilled Cheese"} ]'),  
( 'Batman', 1000,  
  '<address><location city="Gotham City"/><planet>Earth</planet></address>',  
  ' [{"food": "Oatmeal"}, {"food": "Grilled Cheese"} ]'),  
( 'Black Widow', 800,  
  '<address><location city=""/><planet>Earth</planet></address>',  
  ' [{"food": "Medium Rare Steak"}, {"food": "Raspberry Ice Cream"} ]'),  
( 'Popeye', 199,  
  '<address><location city="Mellieha"/><planet>Earth</planet></address>',  
  ' [{"food": "spinach"} ]');
```

Example Data

```
insert into being_traits values
('Storm',      'stealth', 3),
('Batman',     'stealth', 4),
('Black Widow', 'stealth', 5),
('Popeye',     'stealth', 1),

('Storm',      'charisma', 4),
('Batman',     'charisma', 1),
('Black Widow', 'charisma', 2),
('Popeye',     'charisma', 1),

('Storm',      'ruthlessness', 4),
('Batman',     'ruthlessness', 5),
('Black Widow', 'ruthlessness', 3),
('Popeye',     'ruthlessness', 2);
```

Tool Setup

```
# Assumes a linux environment with podman installed.
```

```
# Download the postgres.sh bash script from moodle.
```

```
./postgres.sh          # prints options if no argument is supplied
```

```
./postgres.sh create   # use 'destroy' to remove previous containers
```

```
./postgres.sh psql     # provides an interactive shell
```

```
./postgres.sh import-sql < data.sql
```

```
./postgres.sh import-sql < query.sql
```

SQL/XML (I)

Query

```
select
  xmlelement(
    name being,
    xmlattributes(power_level as powlvl),
    being_name)
from
  beings;
```

Output (4 rows)

```
<being powlvl="9900">Storm</being>
<being powlvl="1000">Batman</being>
<being powlvl="800">Black Widow</being>
<being powlvl="199">Popeye</being>
```

SQL/XML (I)

Query

```
select
  xmlelement(
    name being,
    xmlattributes(power_level as powlvl),
    being_name) as being
from
  beings;
```

Output (4 rows)

```
<being powlvl="9900">Storm</being>
<being powlvl="1000">Batman</being>
<being powlvl="800">Black Widow</being>
<being powlvl="199">Popeye</being>
```

xmlelement expression creates an XML element from the triple
(name <element name>, [xmlattributes], <data>).

xmlattributes expression creates attributes for an XML element from the argument list
(<data> as <attribute name>, ...).

SQL/JSON (I)

Query

```
select
  json_build_object('name', being_name,
                    'powlvl', power_level)
from
  beings;
```

Output (4 rows)

```
{"name" : "Storm", "powlvl" : 9900}
{"name" : "Batman", "powlvl" : 1000}
{"name" : "Black Widow", "powlvl" : 800}
{"name" : "Popeye", "powlvl" : 199}
```

SQL/JSON (I)

Query

```
select
  json_build_object('name', being_name,
                    'powlvl', power_level)
from
  beings;
```

Output (4 rows)

```
{"name" : "Storm", "powlvl" : 9900}
{"name" : "Batman", "powlvl" : 1000}
{"name" : "Black Widow", "powlvl" : 800}
{"name" : "Popeye", "powlvl" : 199}
```

json_build_object creates a JSON object from the pairs in the argument list
(<key name>, <data>, ...).

Also see **json_build_array** for JSON arrays.

SQL/XML (II)

Query

```
select
  xmlagg(
    xmlelement(
      name being,
      xmlattributes(power_level as powlvl),
      being_name)
    )
from
  beings;
```

Output (1 row)

```
<being powlvl="9900">Storm</being><being
powlvl="1000">Batman</being><being
powlvl="800">Black Widow</being><being
powlvl="199">Popeye</being>
```

Note: aggregated result is not valid XML!

SQL/XML (II)

Query

```
select
  xmlagg(
    xmlelement(
      name being,
      xmlattributes(power_level as powlvl),
      being_name)
    )
from
  beings;
```

Output (1 row)

```
<being powlvl="9900">Storm</being><being
powlvl="1000">Batman</being><being
powlvl="800">Black Widow</being><being
powlvl="199">Popeye</being>
```

Note: aggregated result is not valid XML!

xmlagg an aggregate function that concatenates XML fragments across rows.

SQL/JSON (II.a)

Query

```
select
  json_agg(
    json_build_object(
      'name', being_name,
      'powlvl', power_level))
from
  beings;
```

Output (1 row)

```
[{"name" : "Storm", "powlvl" : 9900},
 {"name" : "Batman", "powlvl" : 1000},
 {"name" : "Black Widow", "powlvl" : 800},
 {"name" : "Popeye", "powlvl" : 199}]
```

SQL/JSON (II.b)

Query

```
select json_build_object('beings',
  (select
    json_agg(
      json_build_object(
        'name', being_name,
        'powlvl', power_level))
    from
      beings));
```

Output (1 row)

```
{"beings" : [{"name" : "Storm", "powlvl" : 9900}, {"name" : "Batman", "powlvl" : 1000}, {"name" : "Black Widow", "powlvl" : 800}, {"name" : "Popeye", "powlvl" : 199}]}
```

Note: array has been named to provide context in contrast to the SQL/JSON example in (II.a).

SQL/JSON (II.b)

Query

```
select json_build_object('beings',
  (select
    json_agg(
      json_build_object(
        'name', being_name,
        'powlvl', power_level))
    from
      beings));
```

json_agg aggregates JSON fragments across rows.

Output (1 row)

```
{"beings" : [{"name" : "Storm", "powlvl" : 9900}, {"name" : "Batman", "powlvl" : 1000}, {"name" : "Black Widow", "powlvl" : 800}, {"name" : "Popeye", "powlvl" : 199}]}
```

Note: array has been named to provide context in contrast to the SQL/JSON example in (II.a).

SQL/XML (III)

Query

```
select
  xmlelement(
    name being,
    xmlforest(
      being_name as name,
      power_level,
      address as whereabouts,
      (select xmlforest(
        powerfoods -> 0 -> 'food' as food,
        powerfoods #> '{1,food}' as food) _)
      as diet))
from beings
where being_name = 'Storm';
```

Output (1 row - formatted)

```
<being>
  <name>Storm</name>
  <power_level>9900</power_level>
  <whereabouts>
    <address>
      <location city=""/>
      <planet>Earth</planet>
    </address>
  </whereabouts>
  <diet>
    <food>"Oatmeal"</food>
    <food>"Grilled Cheese"</food>
  </diet>
</being>
```

SQL/XML (III)

Query

```
select
  xmlelement(
    name being,
    xmlforest(
      being_name as name,
      power_level,
      address as whereabouts,
      (select xmlforest(
        powerfoods -> 0 -> 'food' as food,
        powerfoods #> '{1,food}' as food) _)
      as diet))
from beings
where being_name = 'Storm';
```

Output (1 row - formatted)

```
<being>
  <name>Storm</name>
  <power_level>9900</power_level>
  <whereabouts>
    <address>
      <location city=""/>
      <planet>Earth</planet>
    </address>
  </whereabouts>
  <diet>
    <food>"Oatmeal"</food>
    <food>"Grilled Cheese"</food>
  </diet>
</being>
```

xmlforest expression produces a sequence of XML elements from a list of arguments.

(<data> [as <element name>], ...). If an element name is not provided, the column's name is adopted.

SQL/XML (III)

Query

```
select
  xmlelement(
    name being,
    xmlforest(
      being_name as name,
      power_level,
      address as whereabouts,
      (select xmlforest(
        powerfoods [-> 0 -> 'food' as food,
        powerfoods [#> '{1,food}' as food) _]
      as diet))
  )
from beings
where being_name = 'Storm';
```

Output (1 row - formatted)

```
<being>
  <name>Storm</name>
  <power_level>9900</power_level>
  <whereabouts>
    <address>
      <location city=""/>
      <planet>Earth</planet>
    </address>
  </whereabouts>
  <diet>
    <food>"Oatmeal"</food>
    <food>"Grilled Cheese"</food>
  </diet>
</being>
```

[->] returns n-th element of an array if the argument is an integer, otherwise the element stored at the 'key'.

[#>] enables to apply multiple operations to access nested objects, arrays and values using '{<int|text>, ... }'

SQL/JSON (III)

Query

```
select
  json_build_object(
    'name', being_name,
    'power_level', power_level,
    'whereabouts', json_build_object(
      'city',
        (xpath('/address/location/@city',
              address))[1],
      'planet',
        (xpath('/address/planet/text()',
              address))[1]),
    'diet', powerfoods)
from beings
where being_name = 'Storm';
```

Output (1 row - formatted)

```
{
  "name": "Storm",
  "power_level": 9900,
  "whereabouts": {
    "city": "",
    "planet": "Earth"
  },
  "diet" : [
    {"food": "Oatmeal"},
    {"food": "Grilled Cheese"}
  ]
}
```

SQL/JSON (III)

Query

```
select
  json_build_object(
    'name', being_name,
    'power_level', power_level,
    'whereabouts', json_build_object(
      'city',
      (xpath('/address/location/@city',
            address))[1],
      'planet',
      (xpath('/address/planet/text()',
            address))[1]),
    'diet', powerfoods)
from beings
where being_name = 'Storm';
```

Output (1 row - formatted)

```
{
  "name": "Storm",
  "power_level": 9900,
  "whereabouts": {
    "city": "",
    "planet": "Earth"
  },
  "diet" : [
    {"food": "Oatmeal"},
    {"food": "Grilled Cheese"}
  ]
}
```

xpath(<expression>, <xml data>) returns a list of matching nodes in the XML fragment. Since we only expect a single node in the result list, we can select the only node at index 1 with the squared brackets syntax '[1]'.

SQL/XML (IV)

Query

```
select
  xmlelement(
    name beings,
    xmlagg(btraits.btraits))
from
  (select
    xmlelement(
      name being,
      xmlattributes(
        rhs.being_name as name,
        rhs.power_level as level),
      xmlagg(
        xmlelement(
          name trait,
          xmlattributes(
            trait_code as name,
            trait_score as score)))) btraits
    from being_traits lhs right join beings rhs
      on lhs.being_name = rhs.being_name
    where rhs.power_level >= 1000
    group by rhs.being_name
    order by rhs.being_name) btraits;
```

Output (1 row - formatted)

```
<beings>
  <being name="Batman" level="1000">
    <trait name="stealth" score="4"/>
    <trait name="charisma" score="1"/>
    <trait name="ruthlessness" score="5"/>
  </being>
  <being name="Storm" level="9900">
    <trait name="stealth" score="3"/>
    <trait name="charisma" score="4"/>
    <trait name="ruthlessness" score="4"/>
  </being>
</beings>
```

SQL/XML (IV)

Query

```
select
  xmlelement(
    name beings,
    xmlagg(btraits.btraits))
from
  (select
    xmlelement(
      name being,
      xmlattributes(
        rhs.being_name as name,
        rhs.power_level as level),
      xmlagg(
        xmlelement(
          name trait,
          xmlattributes(
            trait_code as name,
            trait_score as score)))) btraits
    from being_traits lhs right join beings rhs
      on lhs.being_name = rhs.being_name
    where rhs.power_level >= 1000
    group by rhs.being_name
    order by rhs.being_name) btraits;
```

Output (1 row - formatted)

```
<beings>
  <being name="Batman" level="1000">
    <trait name="stealth" score="4"/>
    <trait name="charisma" score="1"/>
    <trait name="ruthlessness" score="5"/>
  </being>
  <being name="Storm" level="9900">
    <trait name="stealth" score="3"/>
    <trait name="charisma" score="4"/>
    <trait name="ruthlessness" score="4"/>
  </being>
</beings>
```

SQL/XML (IV)

Query

```
select
  xmlelement(
    name beings,
    xmlagg(btraits.btraits))
from
  (select
    xmlelement(
      name being,
      xmlattributes(
        rhs.being_name as name,
        rhs.power_level as level),
      xmlagg(
        xmlelement(
          name trait,
          xmlattributes(
            trait_code as name,
            trait_score as score)))) btraits
    from being_traits lhs right join beings rhs
      on lhs.being_name = rhs.being_name
    where rhs.power_level >= 1000
    group by rhs.being_name
    order by rhs.being_name) btraits;
```

Output (1 row - formatted)

```
<beings>
  <being name="Batman" level="1000">
    <trait name="stealth" score="4"/>
    <trait name="charisma" score="1"/>
    <trait name="ruthlessness" score="5"/>
  </being>
  <being name="Storm" level="9900">
    <trait name="stealth" score="3"/>
    <trait name="charisma" score="4"/>
    <trait name="ruthlessness" score="4"/>
  </being>
</beings>
```

SQL/XML (IV)

Query

```
select
  xmlelement(
    name beings,
    xmlagg(btraits.btraits))
from
  (select
    xmlelement(
      name being,
      xmlattributes(
        rhs.being_name as name,
        rhs.power_level as level),
      xmlagg(
        xmlelement(
          name trait,
          xmlattributes(
            trait_code as name,
            trait_score as score)))) btraits
    from being_traits lhs right join beings rhs
      on lhs.being_name = rhs.being_name
    where rhs.power_level >= 1000
    group by rhs.being_name
    order by rhs.being_name) btraits;
```

Output (1 row - formatted)

```
<beings>
  <being name="Batman" level="1000">
    <trait name="stealth" score="4"/>
    <trait name="charisma" score="1"/>
    <trait name="ruthlessness" score="5"/>
  </being>
  <being name="Storm" level="9900">
    <trait name="stealth" score="3"/>
    <trait name="charisma" score="4"/>
    <trait name="ruthlessness" score="4"/>
  </being>
</beings>
```

SQL/JSON (IV)

Query

```
select
  json_build_object(
    'beings', json_agg(btraits.btraits))
from
  (select
    json_build_object(
      'name', lhs.being_name,
      'level', power_level,
      'traits', json_agg(
        json_build_object(
          'name', trait_code,
          'score', trait_score))) btraits
  from being_traits lhs right join beings rhs
    on lhs.being_name = rhs.being_name
 where power_level >= 1000
 group by lhs.being_name, power_level
 order by lhs.being_name) btraits;
```

Output (1 row - formatted)

```
{"beings" : [
  {"name": "Batman",
   "level": 1000,
   "traits": [
     {"name": "stealth", "score": 4},
     {"name": "charisma", "score": 1},
     {"name": "ruthlessness", "score": 5}]
  },
  {"name": "Storm",
   "level": 9900,
   "traits": [
     {"name": "stealth", "score": 3},
     {"name": "charisma", "score": 4},
     {"name": "ruthlessness", "score": 4}]
  }
]}
```

SQL/JSON (IV)

Query

```
select
  json_build_object(
    'beings', json_agg(btraits.btraits))
from
  (select
    json_build_object(
      'name', lhs.being_name,
      'level', power_level,
      'traits', json_agg(
        json_build_object(
          'name', trait_code,
          'score', trait_score))) btraits
  from being_traits lhs right join beings rhs
    on lhs.being_name = rhs.being_name
 where power_level >= 1000
 group by lhs.being_name, power_level
 order by lhs.being_name) btraits;
```

Output (1 row - formatted)

```
{"beings" : [
  {"name": "Batman",
   "level": 1000,
   "traits": [
     {"name": "stealth", "score": 4},
     {"name": "charisma", "score": 1},
     {"name": "ruthlessness", "score": 5}]
  },
  {"name": "Storm",
   "level": 9900,
   "traits": [
     {"name": "stealth", "score": 3},
     {"name": "charisma", "score": 4},
     {"name": "ruthlessness", "score": 4}]
  }
]}
```

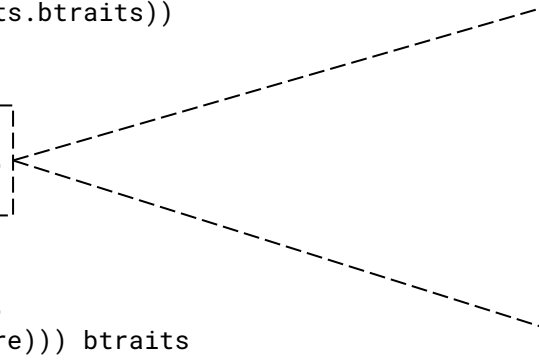

SQL/JSON (IV)

Query

```
select
  json_build_object(
    'beings', json_agg(btraits.btraits))
from
  (select
    json_build_object(
      'name', lhs.being_name,
      'level', power_level,
      'traits', json_agg(
        json_build_object(
          'name', trait_code,
          'score', trait_score))) btraits
  from being_traits lhs right join beings rhs
    on lhs.being_name = rhs.being_name
  where power_level >= 1000
  group by lhs.being_name, power_level
  order by lhs.being_name) btraits;
```

Output (1 row - formatted)

```
{ "beings" : [
  { "name": "Batman",
    "level": 1000,
    "traits": [
      { "name": "stealth", "score": 4},
      { "name": "charisma", "score": 1},
      { "name": "ruthlessness", "score": 5}
    ]
  },
  { "name": "Storm",
    "level": 9900,
    "traits": [
      { "name": "stealth", "score": 3},
      { "name": "charisma", "score": 4},
      { "name": "ruthlessness", "score": 4}
    ]
  }
]
}
```



SQL/JSON (IV)

Query

```
select
  json_build_object(
    'beings', json_agg(btraits.btraits))
from
  (select
    json_build_object(
      'name', lhs.being_name,
      'level', power_level,
      'traits', json_agg(
        json_build_object(
          'name', trait_code,
          'score', trait_score))) btraits
  from being_traits lhs right join beings rhs
    on lhs.being_name = rhs.being_name
  where power_level >= 1000
  group by lhs.being_name, power_level
  order by lhs.being_name) btraits;
```

Output (1 row - formatted)

```
{
  "beings": [
    {
      "name": "Batman",
      "level": 1000,
      "traits": [
        {
          "name": "stealth",
          "score": 4
        },
        {
          "name": "charisma",
          "score": 1
        },
        {
          "name": "ruthlessness",
          "score": 5
        }
      ]
    },
    {
      "name": "Storm",
      "level": 9900,
      "traits": [
        {
          "name": "stealth",
          "score": 3
        },
        {
          "name": "charisma",
          "score": 4
        },
        {
          "name": "ruthlessness",
          "score": 4
        }
      ]
    }
  ]
}
```

References

- Buneman, P. (1997, May). Semistructured data. In Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems (pp. 117-121).
- Dong, X. L., & Srivastava, D. (2013, April). Big data integration. In 2013 IEEE 29th international conference on data engineering (ICDE) (pp. 1245-1248). IEEE.
- Gaillard, M. (2017, July 6). CERN Data Centre passes the 200-petabyte milestone.
<https://home.cern/news/news/computing/cern-data-centre-passes-200-petabyte-milestone>
- Han, J., Haihong, E., Le, G., & Du, J. (2011, October). Survey on NoSQL database. In 2011 6th international conference on pervasive computing and applications (pp. 363-366). IEEE.
- W3C (2016, October 11). Extensible Markup Language (XML). <https://www.w3.org/XML/>
- IETF (2017, December). The JavaScript Object Notation (JSON) Data Interchange Format.
<https://datatracker.ietf.org/doc/html/rfc8259>
- Ackoff, R. L. (1989). "From Data to Wisdom", Journal of Applies Systems Analysis, Volume 16.
- Manning, C. D., Schütze, H. & Raghavan, P. (2008). *Introduction to information retrieval* (Vol. 39, pp. 234-265). Cambridge: Cambridge University Press.
- Ziegler, P., & Dittrich, K. R. (2007). Data integration—problems, approaches, and perspectives. In Conceptual modelling in information systems engineering (pp. 39-58). Berlin, Heidelberg: Springer Berlin Heidelberg.